Journal of Information Security and Applications 48 (2019) 102365

Contents lists available at ScienceDirect

Journal of Information Security and Applications

journal homepage: www.elsevier.com/locate/jisa

Getting to the root of the problem: A detailed comparison of kernel and user level data for dynamic malware analysis

Matthew Nunes*, Pete Burnap, Omer Rana, Philipp Reinecke, Kaelon Lloyd

School of Computer Science & Informatics, Cardiff University, Queen's Buildings, 5 The Parade, Cardiff, CF24 3AA, UK

ARTICLE INFO

Article history:

Keywords:
Dynamic malware analysis
Behavioural malware analysis
API-calls
Machine learning

ABSTRACT

Dynamic malware analysis is fast gaining popularity over static analysis since it is not easily defeated by evasion tactics such as obfuscation and polymorphism. During dynamic analysis it is common practice to capture the system calls that are made to better understand the behaviour of malware. There are several techniques to capture system calls, the most popular of which is a user-level hook. To study the effects of collecting system calls at different privilege levels and viewpoints, we collected data at a process-specific user-level using a virtualised sandbox environment and a system-wide kernel-level using a custom-built kernel driver. We then tested the performance of several state-of-the-art machine learning classifiers on the data. Random Forest was the best performing classifier with an accuracy of 95.2% for the kernel driver and 94.0% at a user-level. The combination of user and kernel level data gave the best classification results with an accuracy of 96.0% for Random Forest. This may seem intuitive but was hitherto not empirically demonstrated. Additionally, we observed that machine learning algorithms trained on data from the user-level tended to use the anti-debug/anti-vm features in malware to distinguish it from benignware. Whereas, when trained on data from our kernel driver, machine learning algorithms seemed to use the differences in the general behaviour of the system to make their prediction, which explains why they complement each other so well. Our results show that capturing data at different privilege levels will affect the classifier's ability to detect malware, with kernel-level providing more utility than user-level for malware classification. Despite this, there exist more established user-level tools than kernel-level tools, suggesting more research effort should be directed at kernel-level. In short, this paper provides the first objective, evidence-based comparison of user and kernel level data for the purposes of malware classification.

1. Introduction

Malware, short for Malicious Software, is the all-encompassing term for unwanted software such as Viruses, Worms, and Trojans. The threat of malware is highlighted by the fact that 350,000 new samples of malware are identified every day [1] — far too many for human analysts to manually analyse, thus motivating research into the automated detection of malware. Malware can be analysed in one of two ways; through static code analysis or dynamic behavioural analysis. *Static code analysis* involves studying the binary file and looking for patterns in its structure that might be indicative of malicious behaviour without ever actually running the binary. *Dynamic behavioural analysis* involves running the binary in a controlled environment, such as an emulated environment, or Virtual Machine (VM), and searching for patterns of Operating System (OS) calls or general system behaviour that are indicative of malicious behaviour. Static analysis has become less effective in recent years due to the fact that malware writers can circumvent detection methods using techniques such as *code obfuscation* and *polymorphism* [2,3]. As a result, behavioural analysis has gained popularity since it actually runs malware in its preferred environment making it harder to evade detection completely.

In order to conduct behavioural analysis, the sample being analysed must be executed in such a way that data relating to the sample's behaviour can be captured while it is running. That data can subsequently be used to train an automated machine learning classifier to distinguish malicious from benign software. One popular mechanism in the literature for understanding malware's behaviour during execution is through capturing the calls made to the OS i.e., system calls. In order to capture this information,

* Corresponding author.
E-mail addresses: nunesma@cardiff.ac.uk (M. Nunes), burnapp@cardiff.ac.uk (P. Burnap), ranaof@cardiff.ac.uk (O. Rana), reineckep@cardiff.ac.uk (P. Reinecke).

https://doi.org/10.1016/j.jisa.2019.102365

a tool must create a *hook* into the OS or monitored process. A hook modifies the standard execution pathway by inserting an additional piece of code into the pathway [4]. This is done in order to interrupt the normal flow of execution that occurs when a process makes a system call and subsequently document the event. There are a number of methods to hook system calls in Windows and these fall into two general categories: those that run in user mode and those that run in kernel mode [4]. Kernel mode is one of the highest privilege levels that can be reached in the computer, whereas user mode is the privilege level that most applications and users operate at. The argument for hooking in user mode is that the code analysing the sample is "closer" to the application being analysed. Whereas, the argument for hooking at kernel mode is that the analysis program resides at a more elevated privilege making it harder for malware to hide from an analysis tool at this level.

The terms *user*' and *kernel* mode are labels assigned to specific Intel x86 *privilege rings* built into their microchips. Privilege rings relate to hardware enforced access control. There are four privilege rings and they range from ring 0 to ring 3 [5]. Windows only uses two of these rings, ring 0 and ring 3. Ring 0 has the highest privileges and is referred to as kernel mode (this is the privilege most drivers run at) by the Windows OS. Ring 3 has the least privileges and is referred to as user mode (and is the level of privileges that most applications run at) [6]. We focus on Windows here because it is still the most targeted OS by malware as reported in [1,7,8].

User-mode hooks tend to only record system/API calls made by a single process since they usually hook one process at a time, whilst kernel-mode hooks are capable of recording calls made by all the running processes at a global, system level. This is an important difference as malware may choose to inject its code into a legitimate process and carry out its activities from there (where it is less likely to be blocked by the firewall). Alternatively, malware could divide its code into a number of independent processes as proposed by Ramilli et al. [9] so that no single process in itself is malicious, but collectively, they succeed in achieving a malicious outcome. Therefore the choice of hooking methodology could affect the quality of the data gained. Another difference between kernel and user level hooks is that each one hooks into a different API. For example, one type of kernel level hook is to hook the System Service Descriptor Table (SSDT) whose calls are similar to those found in the native API, which is mostly undocumented, whilst user mode hooks typically hook the Win32 API which is documented [10]. Although methods in the Win32 API essentially call methods in the native API, there may be some methods in the native API that are unique to it (since it is only supposed to be used by Windows developers) [11]. Likewise, there are some user level methods that do not make calls into the kernel. Therefore, it is of paramount importance that the difference in utility between data collected at each level is objectively studied so that analysts can make an informed choice on which type of data collection method to use. Another factor that could affect the data collected is that due to the differences between the various types of hooking methodologies, malware has to use different techniques to evade each hooking methodology as mentioned by Shaid and Maarof [12]. Consequently, if a piece of malware is focused on avoiding a particular type of hooking methodology, it is likely that any analysts using the same methodology to monitor malware will see a very different picture to those using another methodology. Evasive methods are not uncommon; in fact, one study found evasive behaviour in over 40% of samples [13]. It should also be noted that currently the majority of the existing literature captures *user level* calls as shown in Table A1 in the appendix. This suggests that the literature either believes that user level data has more utility than kernel level data or does not believe there to be a significant difference between user and kernel level data for the purposes of detecting malware (although there are kernel level tools available, they are not as popular as user level tools).

Thus, given the aforementioned evasion concerns and fundamental differences in each class of hooking methodology, the motivation of this paper is to study the differences in data collection at kernel and user level, and consider whether it effects a machine learning method's ability to classify the data. In addition, we provide insights into the utility of the different forms of data collected from a machine when observing potentially malicious behaviour. This is particularly important in the cyber-security domain where the focus tends to be on the data analysis method over the data capturing method. We hypothesise that the features of malware that are used to differentiate it from benignware differ based on the data capturing method used. In order to test our hypothesis, we have created our own *Kernel Driver* that hooks the entire SSDT with the exception of one call. We chose to create our own kernel driver as many of the existing tools that hook the SSDT only monitor calls in a specific category (such as calls relating to the file system or registry) and provide no objective justification as to why they chose the calls they did (if they even make that information available). Therefore, we hook all the calls in the SSDT to ensure we do not miss any subtle details regarding malware behaviour and in order to make an objective recommendation on the most important calls to hook when detecting malware. Our driver is also unique in that it collects the SSDT data at a global system-wide level as opposed to a local process-specific level. In doing this, we expect to determine whether collecting data at a global level assists in detecting malware or is simply adding noise. In order to gather user level data to compare with our driver, we use Cuckoo Sandbox, since it is the most popular malware analysis tool operating at a user level (as shown in Table A1 in the appendix). The data gathered from our driver and Cuckoo is then used to experiment with state of art machine learning techniques to better understand the implications of monitoring machine activity from different perspectives. Alongside the general insights gained from classifying the data, we use feature ranking methods to provide insights concerning the behaviour of malware that is utilised by the classifiers in order to distinguish it. In the interests of transparency and reproduce-ability, we have also made the source code of our kernel driver available at [14] and the data from our experiments available at [15]. The driver can be installed on any system running Windows XP 32-bit and easily be extended to run on Windows 7. In summary, the novel contributions of this paper are the following:

1. We perform the first objective comparison on the effectiveness of kernel and user level calls for the purposes of detecting malware;
2. We compare the usefulness of collecting data for malware detection at a global, system-wide level as opposed to a local, individual process level, providing novel insights into data science methods used within malware analysis
3. We assess the benefits or otherwise of combining kernel and user level data for the purposes of detecting malware;
4. We identify the features contributing to the detection of malware at kernel and user level and the number of features necessary to get similar classification results, providing valuable knowledge on the forms of system behaviour that are indicative of malicious activity;
5. We conduct an extensive survey of dynamic malware analysis tools used or proposed in the literature;
6. We create a driver that hooks all but one call in the SSDT and gathers calls at a global level, which can be used to extend and enhance our work.

The remainder of this paper is structured as follows: Section 2' further describes the various hooking methodologies and the mo-

Fig. 1. System call visualisation.

tivation for this paper. Section 3' describes the various methodologies already employed in the literature to gather kernel calls. Section 4 discusses the experiments that were performed and the environment they were performed in. Section 5 presents and interprets the output from these experiments, and in Section 6, we summarise our work and outline the next steps.

2. Problem definition

2.1. System call structure

In order to understanding how system calls are hooked, it is important to first understand how system calls are structured. Fig. 1 provides an example of the structure of a call tree for a Windows system call. From user mode, a process may call createFileA, createFileW, NtCreateFile, or ZwCreateFile, however, ultimately, they all lead to the NtCreateFile method in the SSDT. In response to a system call being made, the processor must move from Ring 3 (user level) to Ring 0 (kernel level). It does this by issuing the sysenter instruction. Although createFileA has been shown to call NtCreateFile/ZwCreateFile in Fig. 1, strictly speaking, it calls createFileW. However, as they are provided by the same library, they are shown at the same level. From Fig. 1 it can be seen that to get the same information within user mode that is available in kernel mode, more methods need to be hooked. The benefit of hooking in user-mode, however, is that the analysis tool can observe finer details in system calls made. Our aim in this research is to understand if these details are helpful or irrelevant.

2.2. System call hooking

Fig. 2 shows the hooking methods that can be used to intercept system calls organised according to the privilege they hook at.

Fig. 2 shows that there are a number of ways to intercept API-calls using hooks — both at user level and kernel level. Each works in a slightly different way. An *Import Address Table* (IAT) hook modifies a particular structure in a *Portable Executable* (PE) file. The PE file format refers to the structure of executables and DLLs in Windows [16]. IAT hooks exploit a feature of the PE file format, the imports that are listed in a PE file after compilation. An IAT hook modifies the imports so that the import points to an alternative piece of code as opposed to the legitimate function [11,17]. An inline hook refers to when the prologue of a function is replaced in memory with a jump to another piece of code [18]. In Windows, the first five bytes of most functions are the same, therefore, this can be replaced with a jump to an alternative piece of code where the system call can be logged, and then control can be returned back to the original function (after executing the functionality in the first five bytes).

Instrumentation refers to the insertion of additional code into a binary or system for the purpose of monitoring behaviour. *Dynamic instrumentation* implies that this occurs at runtime [19]. SSDT hooks modify a structure in kernel memory known as the System Service Descriptor Table (SSDT). The SSDT is a table of system call addresses that the OS consults to locate a call when it is invoked by a process. An SSDT hook replaces the system call addresses with addresses to alternative code [4,11]. In a *Model Specific Register* (MSR) hook, the value of a specific register is overwritten so that it holds the address of the code performing the hooking. This register is significant as after a system call is made, its value is loaded into the EIP register (which is the register that points to the next instruction to be executed). MSR hooks are frequently employed by Virtual Machine Introspection (VMI) solutions. VMI refers to solutions where the analysis engine resides at the same privilege level as the hypervisor or Virtual Machine Monitor (VMM) [20]. The last method is IRP hooking (a similar goal can be achieved with filter drivers). I/O request packets or IRPs, are used to communicate requests to use I/O to drivers. In *IRP hooking*, a driver intercepts another driver's IRPs [4,11]. Filter drivers are drivers that essentially sit on top of a driver for a device meaning that they receive all the IRPs intended for that driver [21].

There are a number of resources that describe each of the hooking methodologies in much more detail such as [4,11,22]. As can be seen, the way each mechanism intercepts API-calls differs significantly, and each is therefore detected and evaded in a different manner. Furthermore, each mechanism hooks into different APIs (as mentioned previously), depending on whether it is a user-mode or kernel-mode hook. Given all these differences, there is a very real possibility that a tool hooking in user-mode and monitoring a specific process will get different data to a tool monitoring the same process in kernel-mode. This therefore raises the question of which privilege level gathers more beneficial data for the purposes of detecting malware? This is the question this paper attempts to answer.

3. Literature survey

In order to gain a better understanding of the tools used in the literature and the methods that the tools use to gather API-calls, we conducted an extensive review of the literature and noted which tool was used. The results of this are shown in Table A1 in the appendix. Table A1 contains five columns; "Name" which is the name of the tool, "Description" which describes the tool and the hooking methodology it uses, "Kernel Hook" which is marked if the tool employs a hook at kernel level, "User Hook" which is marked if the tool employs a hook at user level, and "Used By" which lists the papers that used that tool. For each tool mentioned in Table A1, if the tool was available online, we tested it in order to understand how it was intercepting API-calls. Where the tool was not available, we used documentation to determine the type of hook being used. To limit the length of the table, Table A1 only

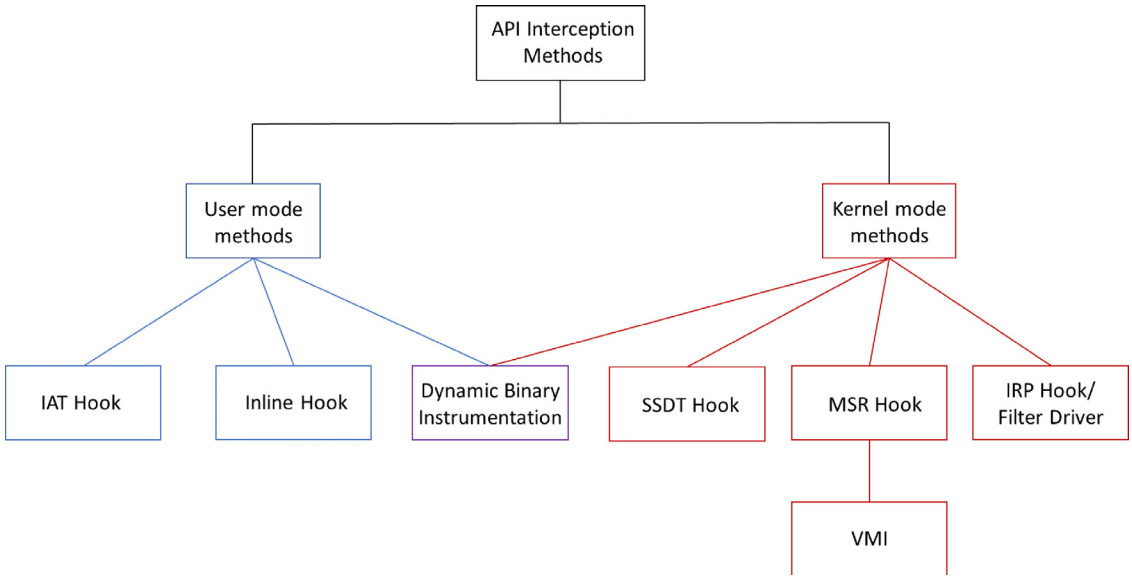

Fig. 2. Hooking methodologies.

contains tools that had been used at least once in the literature (i.e., at least one entry in their "Used By" column).

As can be seen in Table A1, the majority of tools used to gather API calls for the purposes of malware analysis use user level hooks (72%). Currently, the literature suggests that Cuckoo Sandbox is by far the most used tool. However, that does not mean that all papers using Cuckoo collected the same data, as it should be noted that Cuckoo can be enhanced to log additional API calls. Ultimately, all user level tools suffer from the same problem, in that they run at the same privilege levels as the file they are monitoring and are therefore much easier to evade than kernel level tools. In terms of kernel data, there are a number of methods used in the literature to gather data at this level. These can roughly be grouped by the specific hooking method they employ to intercept calls. The four main categories of kernel-mode methods in the literature are: filter drivers, MSR hooks & Virtual Machine Introspection, Dynamic Binary Instrumentation (DBI), and System Service Descriptor Table (SSDT) hooks.

3.1. Filter drivers

Filter drivers do not directly communicate with the hardware but sit on top of lower-level drivers and intercept any data that comes their way. The most well-known tools using filter drivers are Procmon [23] and CaptureBAT [24]. Hăjmăşan et al. [25] take a similar approach to that taken by Procmon and develop a filter driver that registers with Windows callback functions [26] so that it is notified when any changes are made to the registry, file system, or processes. Zhang and Ma [27] take a novel approach by intercepting IRPs in their solution, MBMAS. They then use machine learning to classify sequences of IRPs as malicious or benign. However, the limitation with using filter drivers is that they cannot intercept the same breadth of API-calls that other hooking methodologies can. They focus on the major operations in particular categories (such as file system and registry).

3.2. Model specific register hook

A *Model Specific Register* (MSR) hook essentially hooks the sysenter instruction. More specifically, it involves changing the value of a processor-specific register referred to as the SYSENTER_EIP_MSR register. This register normally holds the address of the next instruction to execute when sysenter is called (which is called every time a system call is made). Therefore if this value is altered, when the sysenter instruction is called, the processor will jump to the address pointed to by the new value in the register (which in this case can point to the analysis engine). Since an MSR hook modifies a processor specific register, developers need to ensure that they modify the registers on each processor (since most systems nowadays contain multiple processors) [6]. There are few examples of an MSR hook being used as a standalone method in the literature. Usually, it is employed in the context of VMI solutions.

VMI refers to tool that operate at the same level as the Hypervisor. This provides benefits such as the ability to monitor a VM without having a large presence on the VM (and thereby making it harder for malware to detect the presence of the analysis engine). The difficulty with monitoring at this level is that a "semantic gap" must be bridged in some way. The semantic gap refers to the fact that when monitoring at the VMM layer, much of the data available is very low level (such as register values). This data is not at a level of granularity that is easy to interpret. Therefore, in order to bridge that, solutions use a number of techniques to convert these values to more abstract values. For example, as mentioned previously, VMI solutions use a variation of the MSR hook whereby instead of placing the address of the analysis solution into the SYSENTER_EIP_MSR register, an invalid value is placed into that register. As a result, every time a system call is made and sysenter is called, a page fault will occur. This will in turn lead to the VMEXIT instruction being called which will pass control to the VMI tool (since it operates at the same level as the hypervisor). The VMI tool must then examine the value of the EAX register in order to find out the system call made. Since monitoring system calls in this manner can have a significant impact on performance, VMI tools usually limit their monitoring to a particular process. To achieve this, the tool must monitor for any changes in the CR3 register. The *CR3 register* contains the base address of the page directory of the currently running process, therefore, if the page directory address of the process of interest is known, then system calls can be filtered to only those emanating from the process of interest.

There are a number of VMI solutions in the literature. TTAnalyze [28] is one of the best known tools employing VMI. TTAnalyze executes malware in an emulated environment (QEMU [29]) as opposed to a virtual one. Unlike virtual environments (where

most instructions are executed on the processor), in emulated environments all instructions are emulated in software. This, they explain, makes it harder for malware to detect that they are not in a real environment since a real system can be mimicked perfectly. However, this comes at the expense of performance, as samples are executed significantly slower. Another well known tool in this domain is *Panorama* [30]. Panorama is built on top of TEMU [31] (the dynamic analysis component of BitBlaze [31] that can perform whole-system instruction-level monitoring), and performs fine-grained taint analysis by monitoring any data touched by the executable being analysed. Its contribution lies in the fine-grained taint tracking it performs, even recording keystrokes among many other things. Ether [32] is a tool in VMI that differs by exploiting Intel VT [33] which enables hardware virtualisation and provides a significant performance boost when running a VM. *Ether* is also particularly focused on not being detectable by malware and, as such, has very little presence on the guest machine. *Osiris* [34] is similar to Ether, however, it manages to perform an even more complete analysis by also monitoring any processes the original process injects its code into. Lengyel et al. [35] propose *DRAKVUF* which focuses more on reducing the presence of an analysis engine from the guest machine as normally there is some code present on the guest to run the process being monitored or help the VMI solution with the analysis. However, DRAKVUF employs a novel method to execute malware using process injection and therefore doesn't require any additional software to be present on the guest. In addition, it monitors calls at both user and kernel level. Pék and Buttyán [36] take a different approach by using invalid opcode exceptions instead of breakpoints to intercept system calls. Invalid opcode exceptions are raised if system calls are disabled when a system call is called. This, they argue, has better performance. In addition, their monitoring solution is not paired with a hypervisor but exploits a vulnerability [37] to virtualise a live system, forgoing the need for a reboot to install the monitoring solution.

While it's clear that significant progress has been made with VMM solutions, there is still a delay overhead incurred from the mechanism (breakpoints/page faults) that is typically used to monitor API-calls. Ether, a well-known tool in this genre, was shown to have approximately a 3000 times slowdown [38]. This, among other things, makes it easier for malware to detect the presence of a monitoring tool. Furthermore, while some solutions have managed to remove much of the presence of the analysis component from the machine being monitored, this has the unfortunate effect of making it even more challenging to bridge the semantic gap.

3.3. Dynamic binary instrumentation (DBI)

Dynamic Binary Instrumentation refers to the analysis of an executable through the injection of additional code into the source or compiled code at runtime. This is usually implemented using a Just-in-Time (JIT) compiler. In DBI, code is executed in basic blocks, and the code at the end of each block is modified so that control is passed to the analysis engine where it can perform a number of checks, such as whether a system call is being executed [39,40]. Two of the most popular frameworks for achieving dynamic instrumentation in Windows are DynamoRIO [39] and Intel Pin [41].

The main limitation in solutions using JIT compilation is Self-Modifying and Self-Checking code (SM-SC) since DBI solutions can be detected by the modifications they make to the code. Therefore, SPiKE [42] was proposed as an improvement to such tools since it uniquely did not use a JIT compiler, but breakpoints in memory. Specifically, it employs "stealth breakpoints" [43], that retain many of the properties of hardware breakpoints, but don't suffer from the limitation that pure hardware breakpoints do of only allowing the user to set between two and four. Through using such breakpoints, it is harder to detect the presence of the monitoring tool and the tool is more immune to SM-SC code. Reportedly, this even brought a performance gain. Polino et al. [40] built their solution, Arancino, on top of Intel Pin which is focused on countering all known anti-instrumentation techniques that are employed by malware to evade detection. They achieve through the use of a number of heuristics.

The problems that solutions in this space suffer from is performance and remaining undetectable by malware. Though [40] make a considerable effort towards improving this, they admit their solution is unlikely to be undetectable.

3.4. SSDT Hooks

This is the method chosen in this paper to monitor API calls at a kernel level. We chose to use an SSDT hook over a filter driver, MSR hook, or DBI tool for a number of reasons. A filter driver tends to obtain the results from calling a system call as opposed to the exact system calls called. While a VMM-layer monitor and DBI tool can suffer from a significant delay due to the manner in which it intercepts system calls, allowing malware to detect a monitor through measuring the delay from performing specific actions. In addition, it can be difficult to deal with SM-SC code with such tools. Furthermore, bridging the semantic gap whilst keeping transparency can be extremely challenging. Ultimately no method is without its limitations (including the SSDT hook), but we chose to use an SSDT hook since it has the most similarities in implementation to a user-level hook (except that it hooks into the undocumented kernel) and the data returned from it is analogous to that returned from a user level hook. Therefore it seems most suitable for the purposes of a comparison. An SSDT hook also has the benefit of not modifying anything on disk (since the SSDT is modified in memory) and therefore leaves a smaller footprint on the analysis machine.

While SSDT hooks have been used previously, they have not had as comprehensive a coverage of calls as ours has. Li et al. [44] employed an SSDT hook to automatically build infection graphs and construct signatures for their system, AGIS (Automatic Generation of Infection Signatures). AGIS then monitors a program to see if it contravenes a security policy and matches a signature. Therefore, it only focuses on calls from a specific process and ignores all other calls. Kirat et al. [45] propose BareBox to counter the problems associated with malware capable of detecting that it is being run in a virtual environment. Barebox runs malware in a real system and is capable of restoring the state of a machine to a previous snapshot within four seconds. Barebox monitors what the authors perceive to be important system calls using an SSDT hook. However, as the number of devices attached to the machine increase, the time it takes Barebox to restore the system to a clean state increases considerably. Grégio et al. [46] propose BehEMOT (Behaviour Evaluation from Malware Observation Tool) which analyses malware in an emulated environment first, then in a real environment if it does not run within the emulated environment. They use an SSDT hook to monitor API calls relating to certain operations. However, by performing analysis on a real environment, BehEMOT suffers a similar problem to Barebox in relation to restoration time. Furthermore, the focus with BehEMOT seems to be producing human-readable and concise reports after each analysis and therefore, only small-scale tests were conducted on a handful of samples.

As mentioned previously, where our solution differs is that previous solutions using SSDT hooks only log calls made to certain API calls by certain processes. Our tool logs all calls (except one) by all processes in order to determine their utility in classification. TEMU is the only tool to offer similar functionality, however, where it differs is that it runs in an emulated environment (which is easier for

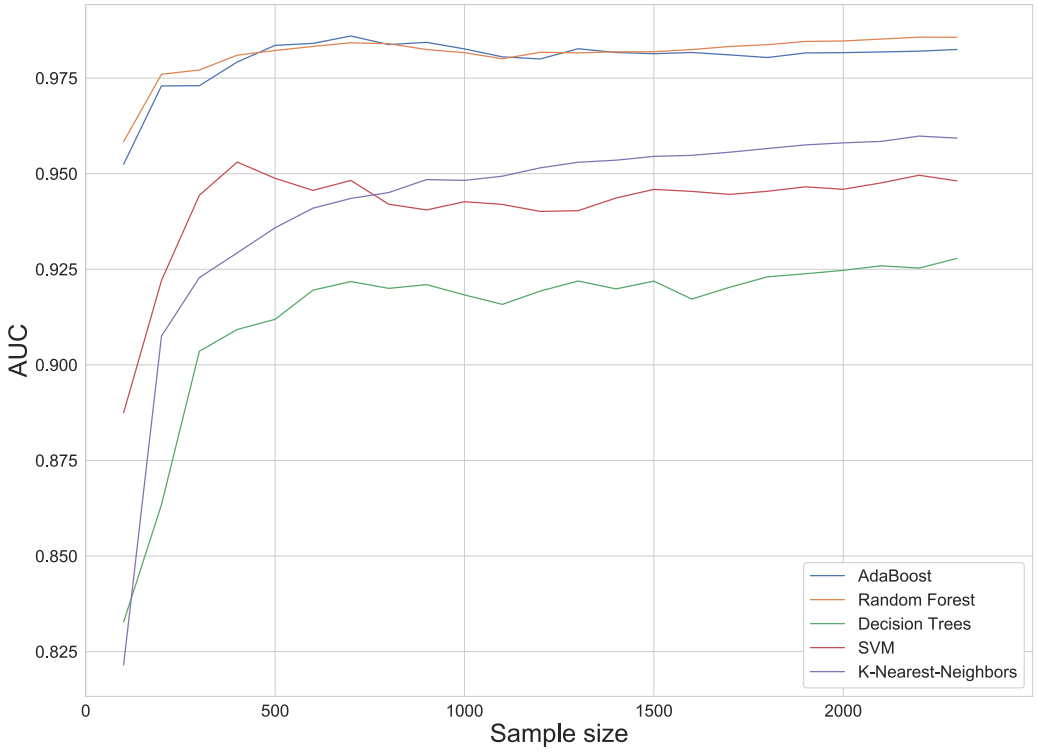

Fig. 3. How the AUC responds as sample size is increased .

malware to detect [47]) and is focused on providing instruction-level details as opposed to high-level system calls.

4. Method & implementation

In order to conduct the experiments required for our study, 2500 malicious samples were obtained from VirusShare [48] and 2500 clean samples were obtained from SourceForge [49] and FileHippo [50]. In order to select an appropriate sample size, we conducted a series of classification experiments (described later) on different sample sizes and monitored the trend in the Receiver Operating Characteristic (ROC) Area Under the Curve (AUC) results (ROC AUC is described later). In these experiments, we varied the sample size from 100 samples up to over 2000 (in increments of 100) and for each sample size, we trained the leading classifiers (using 10-fold-cross-validation) and noted the ROC AUC returned by the classifier. We then plotted the ROC AUC against the sample size and observed when the curve plateaued for each classifier. The results are shown in Fig. 3.

Fig. 3 shows that after 1000 samples, the AUC values almost completely plateau. This suggests that after this point, adding more samples will have an insignificant effect on the classification results. Therefore, we concluded that 2500 samples would be more than enough. In addition, this sample size correlated with the dataset sizes used in the literature [51–54]. The categories of malware in our dataset are shown in Table 1. This information was obtained from VirusTotal [55]. With regards to the clean samples, each was run through VirusTotal to ensure that it was not malicious.

To gather calls made to the SSDT, we wrote a Windows Kernel Driver to hook all but one kernel call in the SSDT since none of the tools available currently provide this. The only call we did not hook, NtContinue, was not hooked due to the fact that hooking it produced critical system errors. Our Kernel driver gathers global data from a system perspective as opposed to simply monitoring calls from a single process introduced into the system. Therefore, the data from the tool can be used to predict whether the machine's state is malicious or not. To gather user level data we chose to use a tool readily available since there are already well established solutions providing this. Specifically, we chose to use the tool most frequently mentioned in the existing literature – Cuckoo (specifically, Cuckoo 2.0.3). Cuckoo is a sandbox capable of performing automated malware analysis.

Table 1
Quantity of each category of malware in our dataset.

Category	Quantity
Trojan	1846
Virus	458
Worm	86
Rootkit	34
Ransomware	23
Adware	22
Keylogger	2
Spyware	2

The experiments were carried out on a virtual machine with Windows XP SP3 installed. We chose to use Windows XP as writing a Kernel driver, particularly one delving in undocumented parts of Windows, is frustratingly challenging. This, however, is made slightly easier in Windows XP due to the fact that it has slowly become more documented through reverse engineering. In addition, all 64 bit systems are backwards compatible with 32 bit binaries [56] and the most commonly prevailing malware samples in the wild are also 32 bit [57] (with not a single 64-bit sample appearing in the top ten most common samples). As of 2016, AVTEST found that 99.69% of malware for Windows was 32 bit [58]. The reason for the popularity of 32 bit malware samples over 64 bit is that its scope is not limited to one architecture. Therefore, given the current prevalence of 32 bit malware, we did not consider that using Windows XP would make our results any less relevant especially since our method could be repeated on other versions of

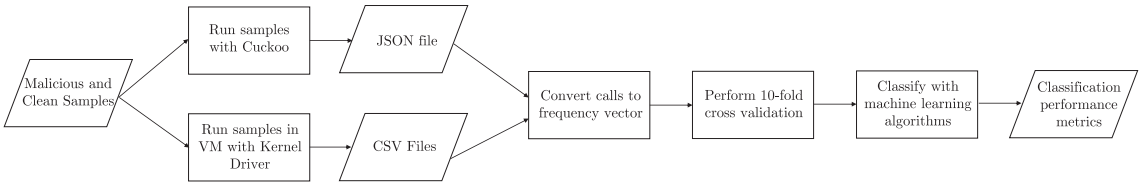

Fig. 4. Workflow diagram of our proposed system's pipeline.

Windows and it would simplify the already challenging engineering task. The host OS was Ubuntu 16.04 and the Hypervisor used was VirtualBox [59]. Both the host and guest machine had a connection to the Internet. In order to ensure fairness and to provide automation, identical sandbox features to Cuckoo (such as simulated human interaction) were implemented for our kernel driver. Fig. 4 shows our system diagram describing the entire experimental process in order to obtain the results.

Our kernel driver creates one CSV file for each system call. A new line is written to each file every time the system call associated with the file is called. After the analysis, a shared folder is used to transfer over the CSV files to the analysis machine. Cuckoo operates in a similar manner however it uses network connections to transfer over analysis files from the VM to the host machine, after which we transfer the JSON file to the analysis machine. We encode the output produced from each of the monitoring tools using a frequency histogram of calls within a two minute period. This feature representation is used to fit a classification model for virus detection.

4.1. Initial experiments' parameters

The transformed data from Cuckoo and the Kernel driver was then classified using a selection of machine learning algorithms provided by scikit-learn [60]. The machine learning algorithms chosen were drawn from the existing literature, as the focus of this research is on the utility of the different views of machine-level actions (user vs kernel) rather than new classification algorithms. The classification algorithms we used were AdaBoost, Decision Tree, Linear SVM, Nearest Neighbours, and Random Forest. The reason we chose these algorithms is that both Decision Trees and SVMs are used widely in the literature [61–66]. Random Forest, while not used as frequently, when used, achieved impressive results [61,65,67,68] as has AdaBoost [61]. In addition, though AdaBoost is an ensemble method like Random Forest, it comes under a different class of ensemble algorithms that use boosting as opposed to bagging (like Random Forest) and therefore may also be capable of strong results. Finally, Nearest Neighbours was chosen due to its simplicity in order to set a baseline. Each of these methods are very well documented, however, briefly, AdaBoost [69] is a collection of weak classifiers (frequently Decision Trees) on which the data is repeatedly fitted with adjusted weights (usually weighting misclassified samples more heavily) until, together, the classifiers produce a suitable classification score or a certain number of iterations are complete. Decision Trees [70] create if-then rules using the training data which they then use to make decisions on unseen data. The K-Nearest Neighbor method picks representative points in each class and when presented with a new observation calculates its proximity to the points and assigns it to whichever is closest. SVMs [71] separate the data by finding the hyperplanes that maximize the distance between the nearest training points in each class. Random Forest [72], like AdaBoost, is a collection of classifiers, and, like AdaBoost, the classifiers are all decision trees. However, AdaBoost tends to employ shallow decision trees while Random Forest tends to use deep decision trees. Random Forest splits the dataset between all the decision trees and then averages the result.

For each classifier, the data was split using 10-fold cross-validation as it is also the standard in this field [54,61,63,73]. It is possible to obtain a number of metrics relating to the performance of the classifiers of which we have chosen to use Area Under the Receiver Operating Characteristic (ROC) Curve (AUC), Accuracy, Precision, and F-Measure since these are the metrics commonly reported in the literature [51,63,64,68,74] and they provide a complete view of the performance of the algorithm without missing out on subtle details (such as the number of false positives). To understand these measures in this context, it is important to define a few basic terms. We interpret True Positives (TP) as malicious samples that are correctly labelled by the classifier as malicious. False Positives (FP) are benign samples that are incorrectly predicted to be malicious. True Negatives (TN) are benign samples that are correctly classified as benign. False Negatives (FN) are malicious samples that are incorrectly classified as benign. With regards to the actual measures used, AUC relates to ROC curves. ROC curves plot True Positive Rate (TPR) against False Positive Rate (FPR). FPR is the fraction of benign samples misclassified as malicious, while TPR represents the proportion of malicious samples correctly classified. A ROC curve shows how these values vary as the classifier's threshold is altered and therefore the AUC is a good measure of a classifier's performance. Accuracy can be described as all the correct predictions (malicious and benign) divided by the total number of predictions. Precision is the number of correctly labelled malware divided by the sum of the correctly labelled malicious samples and the incorrectly labelled clean samples ($\frac{TP}{TP + FP}$). This gives us the proportion of correctly labelled malware in comparison to all samples labelled as malware. Recall is the correctly labelled malicious samples divided by the correctly labelled malicious samples and incorrectly labelled malicious samples ($\frac{TP}{TP + FN}$). This tells us the proportion of malicious samples that are correctly identified. We chose to include precision since false positives are a common issue in malware detection. Recall was not included for brevity and since it can be quickly calculated from F-Measure (which is included) which is the harmonious mean of precision and recall.

In order to confirm whether the differences in classification results were statistically significant or due to randomness, we conducted 10-fold cross-validation 100 times for each classifier. This gave us 1000 AUC values for each classifier. We then checked to see if the 1000 values were normally distributed using Q-Q Plots of the AUC values against a normal distribution. Provided the data was normal, we then performed Welch's t-test [75] in order to determine whether the differences between the classification results were statistically significant or not (with our significance level, α, set to 5% as is commonly used). We used Welch's t-test due to its robustness and widespread recommendation in the literature [76,77].

In addition, in order to gain insight into whether collecting data at a global level is more beneficial for classifying malware, the API calls logged by the kernel driver were reduced to just those coming from the process that was being monitored (and any child processes that it created). Finally, the same data from Cuckoo and our

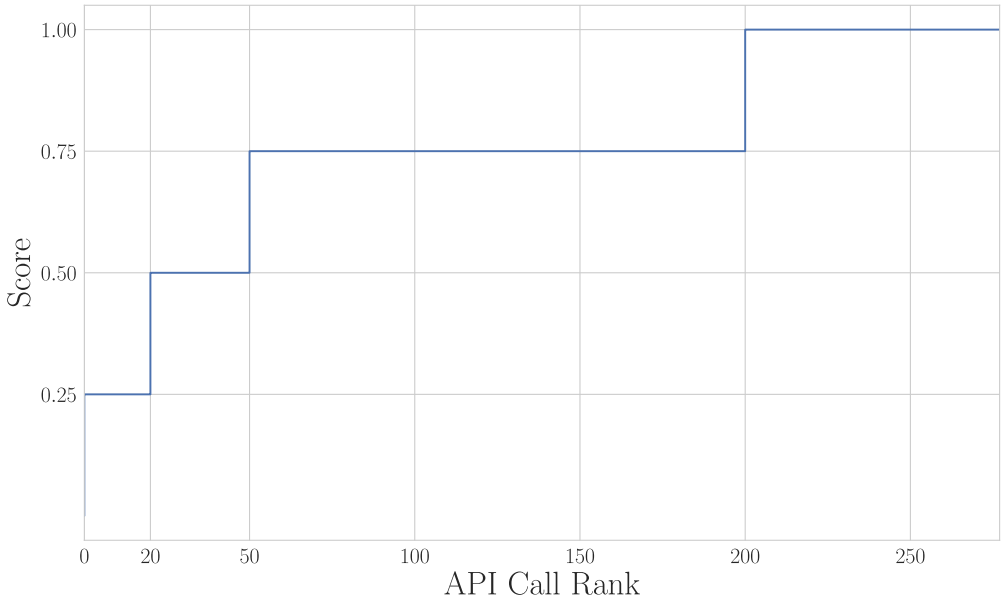

Fig. 5. Example graph of feature ranking mechanism.

Kernel Driver was combined. This was done to see if the combination of user and kernel level data can improve classification results.

4.2. Individual feature ranking

To further understand the data recorded from the kernel and user level, and confirm whether the features being used differ depending on the data collection method used, we ranked features by importance using two metrics for the classifier that had the best results. For the first metric, we put the data from one feature (or API-call) at a time through each classifier and noted the classifier's AUC score in differentiating malicious from clean using only the data from that feature. We refer to this as the independent feature ranking method. This method can give an indication on the strength of individual features. Where it lacks, however, is in its ability to account for the relationship between features. For example, a feature on its own may not be that strong, but when paired with another, may be very strong. Therefore, to account for that, we also rank features using each classifier's in-built feature ranking mechanism (which we refer to as the in-built feature ranking method). This ranking mechanism works in different ways depending on the classifier used. For Decision Trees scikit-learn uses the *Gini importance* as described here [70]. The same is true for Random Forests and AdaBoost since they are composed of a multitude of Decision Trees. The only difference being that as they are composed of multiple Decision Trees, the importance is averaged over each one. Finally, with Linear SVMs, the coefficients assigned to each feature is used to rank them. In the case of K-Nearest Neighbour, there is no in-built feature ranking mechanism, therefore, we do not include it in this measure.

In order to verify that both of the feature ranking methods were selecting features that are optimal, and that the results they produced could be relied on, we created a plot by calculating the AUC using only the top 'x' features where 'x' was gradually increased from 10 by increments of 10 up to the total number of features. In addition, this would show the minimum amount of features necessary to obtain similar classification results

4.3. Complete feature ranking

In order to gain a more consistent but concise view of which features seemed to be assigned a high importance, we created an aggregate measure to rank features across all the classifiers. We applied it to both the in-built and independent feature ranking methods. This will show which features are robust since the previous measure only shows the top ten for the best classifier — which could arguably be skewed in its favour. The aggregate measure was calculated as follows. For each classifier, the features were ranked according to the score they were given by the independent or in-built feature ranking method. Then, the rank was plotted on the x-axis from 0 (the best rank) to the total number of API-calls (the worst rank). On the y-axis was a score from 0 to 1 and at each rank $\frac{1}{\text{number of classifiers}}$ was added to the score. Once this was done, we found the area under the curve and that represented the total strength of the features across all classifiers. This global feature ranking method can be used with any local feature ranking method. Fig. 5 shows an example of this global feature ranking method. In Fig. 5, the feature in question has got the ranks 0, 20, 50, and 200 in the four classifiers it was used with. At each rank, the value has gone up by 1/4 (since there are four classifiers). If a feature was ranked as the most useful feature across all classifiers, its ranks would be 0, 0, 0, and 0, and therefore the area under the curve for it is 1.

5. Results

In this section, we show the results from classifying data collected at a kernel and user level. In addition, in order to further understand the contributing factor to the results for the kernel data, we conduct additional experiments with modified forms of the data. Finally, in order to gain a better understanding of the results, we look at the ten most significant features in order to understand what the machine learning algorithms are using to identify malware

5.1. Initial experiments

The results from classifying data collected using the Kernel Driver at a global level and data collected from Cuckoo are shown in Table 2.

On the whole, the results show that the data from the kernel driver is marginally better for the purposes of differentiating between clean and malicious states regardless of the machine learning algorithm used. The algorithm with the best performance for

Table 2
Comparison of classification results of data from Cuckoo and Kernel driver.

Machine learning algorithm	Kernel driver				Cuckoo			
	AUC	Accuracy	Precision	*F*-measure	AUC	Accuracy	Precision	*F*-measure
AdaBoost	0.983	94.1	0.934	0.941	0.973	91.8	0.911	0.920
Decision Tree	0.944	92.3	0.906	0.925	0.943	87.8	0.918	0.913
Linear SVM	0.945	90.3	0.873	0.906	0.932	86.9	0.835	0.870
Nearest Neighbour	0.964	90.3	0.896	0.903	0.942	86.2	0.877	0.863
Random Forest	0.986	95.2	0.960	0.944	0.984	94.0	0.958	0.942

Table 3
p-values returned from Welch's *T*-Test using AUC values.

Machine learning algorithm	*p*-value
AdaBoost	$1.80 \times e^{-208}$
Decision Tree	$1.41 \times e^{-6}$
Linear SVM	$8.41 \times e^{-78}$
Nearest Neighbour	$9.29 \times e^{-290}$
Random Forest	$2.29 \times e^{-10}$

Table 4
Classification results of data from the Kernel driver focusing on the process under investigation.

Machine learning algorithm	Localised kernel driver			
	AUC	Accuracy (%)	Precision	*F*-measure
AdaBoost	0.962	89.6	0.902	0.891
Decision Tree	0.901	83.8	0.855	0.825
Linear SVM	0.884	82.0	0.893	0.788
Nearest Neighbour	0.934	86.6	0.875	0.858
Random Forest	0.978	92.3	0.944	0.921

Table 5
Classification results from combining Cuckoo and kernel data.

Machine learning algorithm	Cuckoo and kernel driver			
	AUC	Accuracy (%)	Precision	*F*-measure
AdaBoost	0.990	94.9	0.956	0.960
Decision Tree	0.954	92.4	0.924	0.936
Linear SVM	0.952	91.5	0.916	0.915
Nearest Neighbour	0.960	90.3	0.873	0.888
Random Forest	0.990	96.0	0.962	0.942

both Cuckoo and the Kernel driver was Random Forest, obtaining an AUC of 0.986 and 0.984, and an accuracy of 95.2 and 94.0 respectively. We also found that, on average (of 1000 runs), 93% of the samples were given the same label by Random Forest regardless of whether kernel or cuckoo data was used. This shows that while there is agreement on a large number of samples, there are still some samples where data from one was better than the other for classifying malware.

In order to verify whether the difference between the Kernel and Cuckoo classification results are statistically significant and not just occurring by chance, we used Welch's *t*-test on the AUC values as described earlier. A prerequisite for using Welch's *t*-test is that the data must be normally distributed. We verified this using Q-Q plots as shown in Fig. 6.

The Q-Q plots show the distribution of the AUC values and how closely (or otherwise) they relate to the normal distribution (shown as a red line). The plots show that the AUC values barely deviate from the normal distribution. Therefore, Welch's *t*-test would be an appropriate test to observe if the difference between the Kernel and Cuckoo values are statistically significant. Given that the Q-Q plots for the Cuckoo data were very similar, we chose not to show them here for brevity.

In Welch's *t*-test, the null hypothesis is that the means are equal (i.e., H_0: $\mu_1 = \mu_2$), and therefore the alternative hypothesis is that the means are not equal (i.e., H_a: $\mu_1 \neq \mu_2$). We set the threshold α value to be 0.05 as it is an appropriate level for our experimentation. Therefore if the *p*-value returned from performing Welch's *t*-test was less than α, we would reject the null hypothesis. Table 3 shows the results of performing Welch's *t*-test on the AUC values from each classifier.

As Table 3 shows, the *p*-values returned are considerably lower than the threshold, 0.05. Therefore, we reject the null hypothesis that the means of the Kernel and Cuckoo AUC values for each classifier are the same. This shows that, at a significance level of 0.05, the difference between the kernel and Cuckoo results are statistically significant and not just due to chance.

Therefore, from the results in Table 2, we can conclude that data collected at the kernel level produces better classification results than that collected at a user level, however, it is unclear whether this is because the data collected at a kernel level was at a higher privilege and hooking a different API, or because the data was collected on a global scale of all running processes allowing us to see everything happening on the machine. In order to clarify whether collecting the data at a global level assisted or harmed the classification process, we limited the kernel data collected to that of the data produced by the process being analysed and any processes it created. The results from this are shown in Table 4.

From Table 4, it can be seen that the classification results have decreased when collecting data from the kernel driver at a local, process-specific, level. For example, with Random Forest the AUC has decreased from 0.986 to 0.978 and the accuracy from 95.2% to 92.3%. In addition, the differences between global and local kernel data were also found to be statistically significant. Therefore, it is evident that collecting data at a kernel level is not the only contributing factor to the improved classification results over user level, data must also be collected at a global-level in order to obtain better classification results. It is also interesting to note that, at a significance level of 0.05, the classification results from localised Kernel data are statistically significantly lower than the Cuckoo results as well. This shows that if data is going to be collected at a process-specific level, user-level hooks provide more value since they will also observe many of the process' interactions that did not reach the kernel. In addition, this shows that simply collecting at a kernel privilege is not enough. The scope of the collection (local vs global) is also important. It may be possible to improve the localised Kernel results slightly by attempting to detect when malware injects its payload into benign software and runs it from there. However, that data would be captured by a global Kernel capture and therefore we wouldn't expect the results to improve beyond the global kernel results.

Since limiting the data from the kernel driver did not improve results, and given that Cuckoo and the Kernel Driver seemed to fail on different samples, we combined the data from Cuckoo and the Kernel driver in order to see whether classification results are improved by a combination of data from both levels. The results of this are also shown in Table 5.

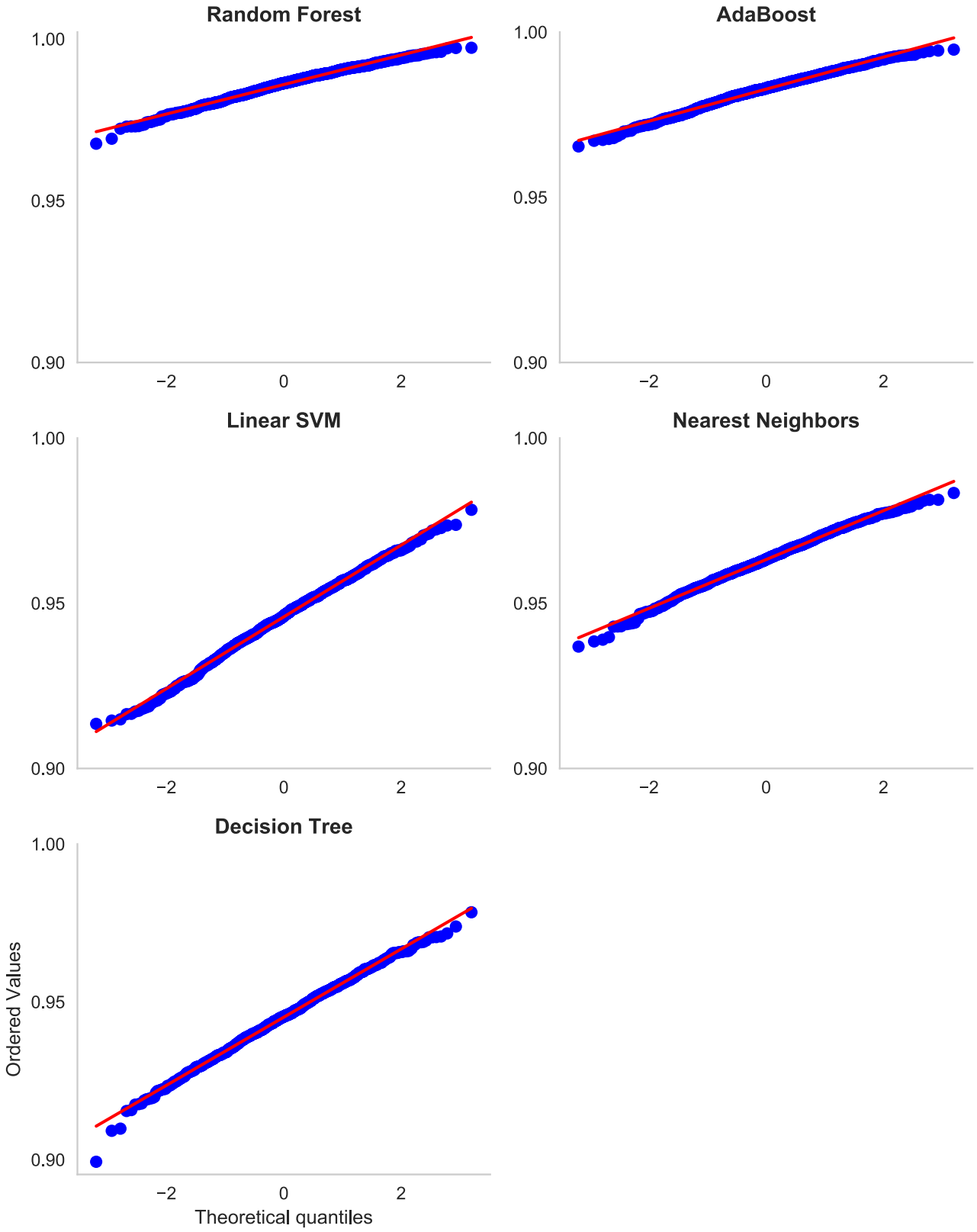

Fig. 6. Q-Q Plots of AUC values from Kernel data.

Table 6
Top ten features using independent feature ranking with Random Forest.

Cuckoo	Kernel driver
GetSystemMetrics	NtQueryDebugFilterState
LoadResource	NtEnumerateKey
FindResourceExW	NtQueryFullAttributesFile
NtQueryInformationFile	NtReleaseSemaphore
SetFileTime	NtEnumerateValueKey
NtUnmapViewOfSection	NtReadVirtualMemory
NtOpenSection	NtSetInformationProcess
NtWriteFile	NtSetValueKey
FindResourceA	NtOpenEvent
CreateDirectoryW	NtNotifyChangeKey

Table 7
Top ten features using in-built feature ranking with Random Forest.

Cuckoo	Kernel driver
GetSystemMetrics	NtWriteFile
FindResourceA	NtFlushVirtualMemory
LdrGetProcedureAddress	NtReadFile
LoadResource	NtUnlockFile
NtReadFile	NtOpenMutant
NtQueryInformationFile	NtLockFile
SetFileTime	NtNotifyChangeDirectoryFile
GetFileAttributesW	NtOpenEvent
NtOpenSection	NtDeleteAtom
NtUnmapViewOfSection	NtQueryValueKey

Table 5 shows that combining data from both tools produces classification results that are slightly stronger for the purposes of malware classification with an AUC of 0.990 for both AdaBoost and Random Forest. The only classifier with reduced results was K-Nearest-Neighbours suggesting that it struggles to classify data beyond a certain number of dimensions. Again, as with all the data, the differences shown in this table (improvements or otherwise) are statistically significant. Therefore, this further validates the claim that there is a difference in the data from Cuckoo and the Kernel Driver and that they fail on different samples since the results would not have improved had this not been the case.

5.2. Individual feature ranking

In order to further understand and confirm the differences between the data gathered by Cuckoo and the Kernel Driver, we compare the top ten features using both feature selection methods (described in Section 4.2 – Individual Feature Ranking)for Random Forest since it is the best performing algorithm. Table 6 compares the top ten features (in order of score) using the independent feature ranking method for Cuckoo and the Kernel driver. Table 7 shows the same, but using the in-built feature selection method. The feature importance is shown only for Random Forest since it had the best performance. While it would have been ideal to show a comparison of all the calls rather than simply the top ten, due to the limitations of space, we have chosen to restrict it to ten. If the data being used by the machine learning algorithms is the same and therefore the difference in results is due to some other factor, we would expect the top ten features to be identical or near identical.

From Table 6, we can see that the data collected from Cuckoo and the Kernel do not have any features in common in the top ten for the independent feature ranking method. This suggests that both views used very different indicators to distinguish malware. In terms of the actual methods in the top ten for each tool, the kernel driver contains relatively generic calls relating to the registry, threading, memory, events, and processes. Whereas Cuckoo contains some highly specific calls such as SetFileTime (to set MAC (modify, access, and create) times on a file) and GetSystemMetrics (to get information about the system). The presence of SetFileTime is not surprising as it is often used by malware to conceal conceal its accesses of a file (and thereby conceal its malicious activity) [78]. GetSystemMetrics is used by malware to evaluate whether it is running in a virtual environment or a real one (since virtual machines tend to have low memory and storage). NtUnmapViewOfSection (and NtOpenSection) is also used to evade detection as malware can use it to replace the code of a legitimate process in memory with its code so that the legitimate process runs its code. This could be the reason why the kernel driver monitoring at a global level performed better than Cuckoo monitoring at a local level as it was able to capture this behaviour better. The top ten also includes some methods relating to resources (LoadResource and FindResourceExW), malware tends to hide its payload inside the resource section of a PE file, and therefore these methods would be used to extract it into memory. What is also noticeable in Cuckoo's top ten is a mix of calls from the native API (usually starting with Nt) and the Win32 API. An example of that is NtQueryInformationFile, used to obtain information about a file. The reason for malware using this method over an equivalent Win32 call is that it provides more information. It's clear that the vast majority of features favoured by classifiers to distinguish malware in the Cuckoo data are the evasive features of malware, whereas the Kernel Driver uses differences in the general behaviour of malware to distinguish it from benignware.

Table 8
Top ten features using in-built feature ranking with Random Forest.

Cuckoo	Kernel driver
GetSystemMetrics	NtReleaseSemaphore
NtQueryInformationFile	NtLockFile
LoadResource	NtUnlockFile
RegQueryValueExW	NtEnumerateKey
NtUnmapViewOfSection	NtWriteFile
NtDuplicateObject	NtOpenMutant
RegOpenKeyExW	NtReadFile
RegCloseKey	NtOpenThreadToken
NtOpenSection	NtReplyWaitReceivePortEx
NtWriteFile	NtQueryVirtualMemory

Much of our discussion about the top ten features in Cuckoo for Table 6 also applies to the features of Cuckoo in Table 7. However, unlike Table 6, there is one method in common between the kernel and cuckoo features, NtReadFile. This suggests that this feature is important regardless of the perspective from which data is being gathered. Another interesting observation is that there are seven methods in common between Cuckoo's independent (Table 6) and inbuilt feature ranking (Table 8). This suggests that many of the contributing features in Cuckoo's case can be used alone to detect malware (which is worth considering when selecting feature representation methods). Due to this, many of the observations made about Cuckoo's top ten in Table 6 apply here (such as Cuckoo focusing more on malware's evasive behaviour over its general behaviour). Aside from this, Cuckoo's top ten in Table 7 also contains LdrGetProcedureAddress. This is important as it can be used by malware to evade static analysis and dynamic heuristic analysis by loading all the routines it needs at runtime and therefore malware can achieve all that it intends to with only that method linked at compile time.

On the Kernel side, there is one method in common between the inbuilt and independent feature ranking method, NtOpenEvent. This is no surprise as this method can be used to interact with Windows Events which malware could use to ensure it is run every day, for example. In general, the top tens for the kernel data

for both tables are more focused on the differences in general process behaviour between malware and benignware. There are fewer methods directly related to specific behaviour exhibited by malware, however, there are a few exceptions. In the independent feature ranking for Kernel data shown in Table 6, there is the method NtSetInformationProcess, which has been known to be used by malware to disable Data Execution Prevention (DEP). DEP is a protection in memory which prevents malware from running code in non-executable sections of memory [79]. Another method in the top ten likely to be related to malware is NtNotifyChangeKey. This is used by a process to ask Windows to notify it whenever any changes are made to the registry. This could be used by malware to monitor what is being done on the system or even prevent any changes to the keys that it created.

The top ten for the Kernel data using the inbuilt feature ranking method (shown in Table 7) also reflects this. As with the previous table, there are some unusual methods in the top ten features for the Kernel data; for example, NtNotifyChangeDirectoryFile, a completely undocumented method. This method is used by a process to ask Windows to notify it when any changes occur in a directory, therefore, malware may be using it to simply monitor system activity and protect itself or to attach itself to any file moves. However, another likely reason is that this method is responsible for a publicised vulnerability [80] that could be used to expose parts of kernel memory and defeat Address Space Layout Randomisation (ASLR). NtNotifyChangeDirectoryFile is not the only undocumented method in the top ten; NtDeleteAtom and NtOpenMutant are also completely undocumented by Windows. This could explain why the Kernel data was able to better distinguish malware from benignware as it is able to capture behaviour that cannot be captured at user level. Aside from that, the differences in general process behaviour are being used to detect malware.

Tables 6 and 7 demonstrate that Random Forest, when trained on data from Cuckoo and the Kernel Driver, utilises different behavioural aspects when identifying if a file is malicious or not. While Cuckoo and our kernel driver generally monitor equivalent calls, the fact that the observed rankings are different suggests that the scope (local or global) of the calls is an important factor. Another contributing factor could be that malware evades or detects the inline API-hooking technique used by Cuckoo but not the Kernel hooking method employed by our driver (since it requires a more sophisticated approach to evade).

To confirm the correctness of both of the feature ranking methods, we performed some simple feature reduction (described in "Section 4 - Method & Implementation") using our feature ranking methods. The results of this are shown in the Figs. 7 and 8. We created these graphs for both the data from the kernel driver, and the data from the Cuckoo driver. However, since the graphs were a very similar shape, for brevity's sake, we have only shown the graphs for the data from the Kernel driver.

For most of the plots in Figs. 7 and 8 the AUC is at its lowest with just ten features, however, as the number of features that the machine learning algorithms use increases, the AUC increases until it reaches its peak at around 50 features after which the introduction of new features simply adds noise, thereby reducing or not contributing to the difference in the AUC. This highlights that the feature ranking method seems to be able to decipher which features are important. In addition, it shows that, in most cases, no more than 50 API-calls need to be hooked for similar results.

5.3. Complete feature ranking

Finally, we applied the global feature ranking metric we created (described in "Section 4.3 – Complete Feature Ranking") to get a concise yet comprehensive view of the features of malware that were consistently considered important by all classifiers. The results from applying the global feature ranking for both the inbuilt and independent feature selection methods are shown in Tables 8 and 9.

Table 9
Top ten features using in-built feature selection considering all classifiers.

Cuckoo	Kernel driver
NtOpenSection	NtFlushVirtualMemory
InternetCloseHandle	NtOpenMutant
LoadResource	NtFilterToken
SetUnhandledExceptionFilter	NtUnlockFile
SetFileTime	NtAccessCheckByTypeAndAuditAlarm
LdrLoadDll	NtQueryVirtualMemory
CreateActCtxW	NtDeleteAtom
getaddrinfo	NtWriteFile
LdrGetDllHandle	NtReadFile
LdrGetProcedureAddress	NtCompleteConnectPort

From these tables we can ascertain which features perform best across all the classifiers that we used. This gives us a clearer picture of which features are extremely strong when it comes to differentiating malware from cleanware. With regards to the Cuckoo data, we see in Table 8 some of the features used to evade detection that we have seen before (GetSystemMetrics, NtUnmapViewOfSection, and NtOpenSection). There are also resource related methods (LoadResource) and the native API method (NtQueryInformationFile) we encountered previously. Of the new methods, NtDuplicateObject is interesting because it is used by malware to evade anti-virus heuristics, as anti-viruses would expect malware to call the more commonly used DuplicateHandle to duplicate a process handle to kill or inject into it and would therefore be less likely to flag a call to NtDuplicateObject as suspicious [81]. From this we can conclude that Cuckoo's top ten in Table 8 contains a mix of evasive, potentially malicious, and general methods.

In contrast, Cuckoo's top ten in Table 9 has more emphasis on the evasive behaviour of malware. For example, LdrLoadDll, LdrGetDllHandle and LdrGetProcedureAddress are in the top ten and are known to be used by malware to load DLLs dynamically in order to import methods from them. This can be used to avoid being detected by IAT hooks. In addition, the method SetUnhandledExceptionFilter in the Cuckoo top ten, is also used as an anti-debugging trick by malware as this method is used to specify a function to be called in the event of an exception occurring that is not handled by any exception handler. However, the function specified will only be called if the process that raised the exception is not being debugged. Therefore, malware can register a function to deliver its payload and then throw an exception, and if the process is being debugged, that function will not be called, and hence the malware will not display its malicious behaviour. SetFileTime, which has been described previously, is also used to curb suspicions. Finally, NtOpenSection, as mentioned previously, can be used to embed malicious code in a benign process. Therefore, as can be seen, much of the top ten for Cuckoo in Table 9 utilise the evasive behaviour of malware to detect it.

On the Kernel side, each table contains methods from a wide range of categories (such as file-system, threading, networking etc.), making it more general than the top ten kernel calls in the Cuckoo data. While many of the methods in these tables are likely to be used by malware, they are not used solely by malware (as would be expected from a tool monitoring at a global level). On the whole, it can be seen that with the Cuckoo data, malware is detected through the techniques it uses to detect a monitoring or virtual environment, whereas, with the data from the Kernel Driver, malware is differentiated from cleanware through how its general behaviour differs from the norm.

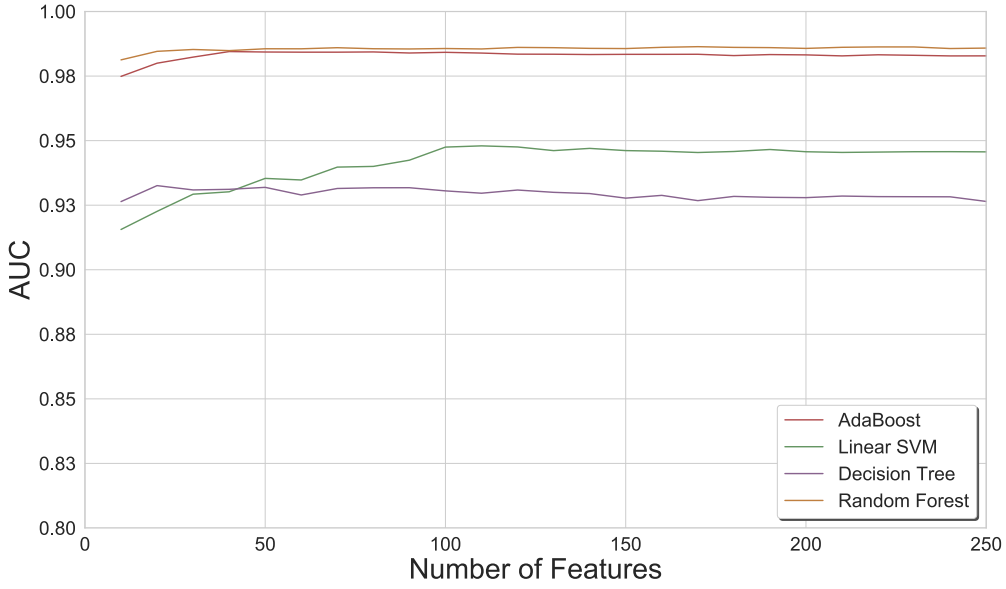

Fig. 7. Feature selection using inbuilt feature selection method.

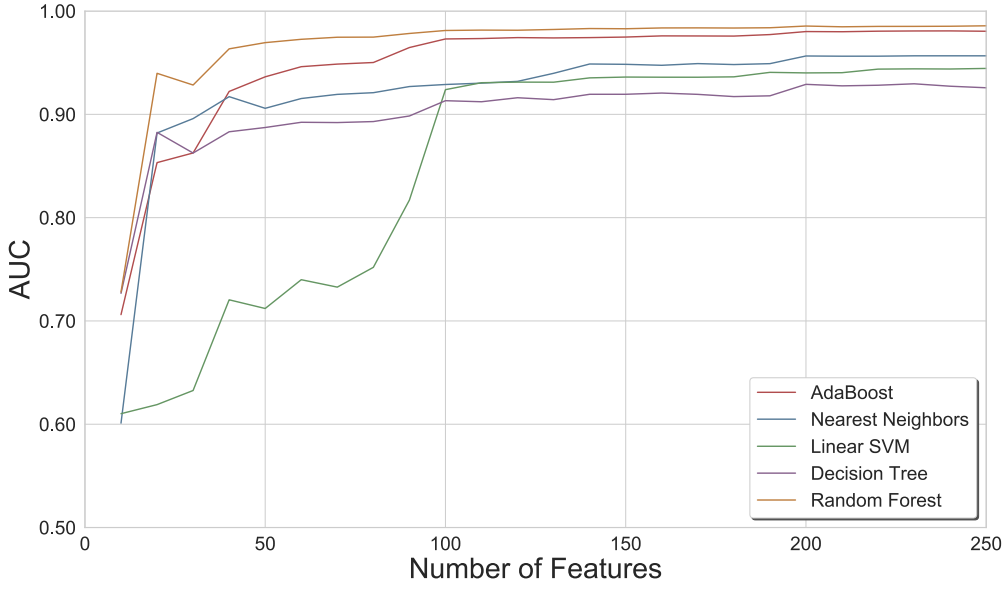

Fig. 8. Feature selection using independent feature selection method.

6. Conclusion

Motivated by a hypothesis that kernel level API calls and user level API calls do not produce the same classification results, we conducted experiments to understand the differences by collecting data at different privilege levels within the same Operating System. We collected data at a user level using Cuckoo, and at the kernel level using a custom made Kernel driver since there are no existing tools that hook all the calls in the SSDT on a global scale. The data collected was classified using several state-of-the-art machine learning algorithms to determine whether collecting data at different levels altered classification results. The results showed kernel data to be statistically significantly better for all classification algorithms despite the fact that user level methods are significantly more popular in the literature. Random Forest performed the best with an accuracy of 94.0% for Cuckoo and 95.2% for the Kernel Driver. In addition, by limiting the kernel data to that produced by the process under observation (and its subprocesses), we found that the classification results reduced suggesting that the collection of data at a global, system-wide level aided the classification process. Our strongest classification results were observed by combining the data from Cuckoo (user level) with that from our Kernel driver; achieving an AUC of 0.990 and accuracy of 96.0% for Random Forest.

In order to understand why the differences in data collection methods had contributed to the different classification results, we performed feature ranking for Random Forest and collectively for all classifiers used, and found that the features focused on by classifiers differed significantly from the data used. The main observation from this was that monitoring on a process specific level as Cuckoo does caused the machine learning algorithm to detect malware using its evasive properties. Whereas, when trained on data obtained from monitoring at a global, kernel level, the machine learning algorithm used the more general behaviour of the malware (and processes in general) to distinguish it from cleanware. The differences resulting from collecting data at different privilege levels highlighted the benefit gained from collecting data at a kernel level (or both levels) in order to detect malware and

the importance of the literature carefully detailing the data collection method that has been used since the results are affected by it. To assist with this, we have documented many of the dynamic malware analysis tools in Table A1 in the appendices of this paper. Table A1 shows that while there exists a plethora of well established tools for collecting data at a user level, there are only a handful of established tools to collect data at a kernel level, and fewer still that are freely available. While the driver we have written is specific to Windows XP, the main contributions of this paper (a comparison of user and kernel level calls) will apply to future releases of Windows. In conclusion, this paper provides the first objective, evidence-based comparison of kernel level and user level data for the purposes of malware classification. In future we hope to do an in-depth analysis into the implications of the differences in the representative features of malware with kernel and user data.

Funding

This work has been supported by the Engineering and Physical Sciences Research Council [project no. 1657416].

Declaration of Competing Interest

We would like to reiterate that we have no conflicts of interest to disclose.

Acknowledgments

We would also like to thank VirusShare and VirusTotal for providing us with samples and information regarding malware.

Appendix A. Tools used in the literature to gather API-calls

Table A1

Name	Description	Kernel hook	User hook	Used by
API Monitor [82]	Capable of hooking every method in the Windows API		x	[63,83–85]
APIMon [86]	Uses EasyHook [87] to perform inline hooking on all user-level APIs		x	[88]
Buster Sandbox Analyser [89]	Not documented how it gathers API calls. Monitors specific categories of calls.		x	[53,90]
CaptureBAT [24]	Uses filter drivers	x		[91,92]
Cuckoo Sandbox [93]	Leading open-source dynamic malware analysis system [93]. Uses inline hook to hook certain categories of Windows API calls [94]		x	[51,52,54,67,68,73,95–127]
CWSandbox [128]	Uses in-line code hooks to record calls in specific categories [128]		x	[129–134]
Deviare [135]	Hooking engine that hooks entire Win32 API and is also integrate-able with many programming languages		x	[65]
Ether [32]	VMI solution focused on being undetectable by malware (known for achieving good transparency). Utilises Xen hypervisor and Intel VT [33] to provide hardware virtualization		x	[53,64,136,137]
HookMe	Uses Microsoft's Detours [18] to perform in line hooking		x	[61,138]
Malpimp [139]	Based on pydbg (pure Python debugger)		x	[140]
Micro analysis System (MicS) [141]	Executes in a real (not virtual) environment and uses IAT hooking		x	[142]
NtTrace [143]	Tool that uses inline hooking to hook ntdll.dll		x	[144]
Osiris [34]	VMI solution using a modified version of QEMU [29]. Also provides a simulated network environment. Monitors specific set of user and kernel level calls	x	x	[64]
StraceNT [145]	Inspired by strace on Linux. Uses IAT hooking to hook all user-level APIs		x	[146–148]
Sysinternals Process Monitor [23]	Gathers data using a kernel driver (file system filter driver) [6]	x		[51,91,149–153]
TEMU [154]	Extensible complete-system, fine-grained analysis platform capable of monitoring any call	x	x	[30,155,156]
TTAnalyze (used in Anubis (Analysis of unknown binaries) Sandbox [157] [158])	Uses QEMU [29] to perform software emulation. Monitors specific categories of API calls through JIT compilation [28]	x	x	[62,159–163]
WinAPIOverride [164]	Free tool to monitor all user-level Windows API calls made by processes		x	[165,166]

References

[1] AVTEST. The AV-TEST Security Report 2016/17. Tech. Rep.; 2017. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2015-2016.pdf.

[2] Liu J, Wang Y, Wang Y. The similarity analysis of malicious software. In: 2016 IEEE first international conference on data science in cyberspace (DSC); 2016. p. 161–8. doi:10.1109/DSC.2016.12.

[3] Moser A, Kruegel C, Kirda E. Limits of static analysis for malware detection. In: Twenty-third annual computer security applications conference (ACSAC 2007); 2007. p. 421–30. doi:10.1109/ACSAC.2007.21.

[4] Rudd EM, Rozsa A, Günther M, Boult TE. A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions. IEEE Commun Surv Tutor 2017;19(2):1145–72. doi:10.1109/COMST.2016.2636078.

[5] Schroeder MD, Saltzer JH. A hardware architecture for implementing protection rings. Commun ACM 1972;15(3):157–70. doi:10.1145/361268.361275.

[6] Russinovich ME, Solomon DA, Ionescu A. Windows internals part 1. 6th ed; 2012. ISBN 978-0-7356-4873-9.

[7] Garnaeva M, Sinitsyn F, Namestnikov Y, Makrushin D, Liskin A. Overall statistics for 2016; https://kasperskycontenthub.com/securelist/files/2016/12/Kaspersky_Security_Bulletin_2016_Statistics_ENG.pdf.

[8] Symantec. Internet security threat report 21. https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf.

[9] Ramilli M, Bishop M, Sun S. Multiprocess malware. In: Proceedings of the 2011 6th international conference on malicious and unwanted software. MALWARE '11. Washington, DC, USA: IEEE Computer Society; 2011. p. 8–13. ISBN 978-1-4673-0031-5. doi:10.1109/MALWARE.2011.6112320.

[10] Nebbett G. Windows NT/2000 native API reference. Thousand Oaks, CA, USA: New Riders Publishing; 2000. ISBN 1578701996.

[11] Blunden B. The rootkit arsenal: escape and evasion in the dark corners of the system. 2nd ed. USA: Jones and Bartlett Publishers, Inc.; 2012. 144962636X, 9781449626365

[12] Shaid SZM, Maarof MA. In memory detection of windows api call hooking technique. In: 2015 International conference on computer, communications, and control technology (I4CT); 2015. p. 294–8. doi:10.1109/I4CT.2015.7219584.

[13] Chen X, Andersen J, Mao ZM, Bailey M, Nazario J. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: 2008 IEEE international conference on dependable systems and networks with FTCS and DCC (DSN); 2008. p. 177–86. doi:10.1109/DSN.2008.4630086.

[14] Nunes M. Matthewnunes/kernelssdtdriver: kernel driver (with localisation); 2018. doi:10.5281/zenodo.1169136.

[15] Nunes M. Dynamic malware analysis kernel and user-level calls; 2018. doi:10.17035/d.2019.0082395337.

[16] Pietrek M. Inside windows-an in-depth look into the win32 portable executable file format. MSDN Mag 2002;17(2).

[17] Leitch J. Iat Hooking Revisited; 2011.

[18] Hunt G, Brubacher D. Detours: binary interception of win32 functions. In: 3rd usenix windows nt symposium. 1999.

[19] skape. Dynamic binary instrumentation. Uninformedorg 2007;7.

[20] Garfinkel T, Rosenblum M. A virtual machine introspection based architecture for intrusion detection. In: Proc. network and distributed systems security symposium, 3; 2003. p. 191–206.

[21] Viscarola P, Mason WA. Windows NT device driver development. 1st ed. Thousand Oaks, CA, USA: New Riders Publishing; 1998. ISBN 1578700582.

[22] Hoglund G, Butler J. Rootkits: subverting the windows kernel. Addison-Wesley Professional; 2005. ISBN 0321294319.

[23] Russinovich ME. Process monitor — windows sysinternals | microsoft docs. https://docs.microsoft.com/en-gb/sysinternals/downloads/procmon; Visited on 2017-07-27.

[24] The Honeynet Project. http://old.honeynet.org/index.html Visited on 2017-07-26;

[25] Hăjmăşan G, Mondoc A, Creţ O. Dynamic behavior evaluation for malware detection. In: 2017 5th International symposium on digital forensic and security (ISDFS); 2017. p. 1–6. doi:10.1109/ISDFS.2017.7916495.

[26] Callback Objects | Microsoft Docs. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/callback-objects Visited on 2017-07-26;

[27] Zhang F, Ma Y. Using irp with a novel artificial immune algorithm for windows malicious executables detection. In: 2016 International conference on progress in informatics and computing (PIC); 2016. p. 610–16. doi:10.1109/PIC.2016.7949573.

[28] Bayer U. TTAnalyze: a tool for analyzing malware; 2005. http://old.iseclab.org/people/ulli/TTAnalyze_A_Tool_for_Analyzing_Malware.pdf

[29] Bellard F. Qemu, a fast and portable dynamic translator. In: Proceedings of the annual conference on USENIX annual technical conference. ATEC '05; Berkeley, CA, USA: USENIX Association; 2005, p. 41–41http://dl.acm.org/citation.cfm?id=1247360.1247401.

[30] Yin H, Song D, Egele M, Kruegel C, Kirda E. Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM conference on computer and communications security. CCS '07. New York, NY, USA: ACM; 2007. p. 116–27. ISBN 978-1-59593-703-2. doi:10.1145/1315245.1315261.

[31] Song D, Brumley D, Yin H, Caballero J, Jager I, Kang MG, et al. Bitblaze: a new approach to computer security via binary analysis. In: Proceedings of the 4th international conference on information systems security. ICISS '08. Berlin, Heidelberg: Springer-Verlag; 2008. p. 1–25. ISBN 978-3-540-89861-0. doi:10.1007/978-3-540-89862-7_1.

[32] Dinaburg A, Royal P, Sharif M, Lee W. Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM conference on computer and communications security. CCS '08. New York, NY, USA: ACM; 2008. p. 51–62. ISBN 978-1-59593-810-7. doi:10.1145/1455770.1455779.

[33] Uhlig R, Neiger G, Rodgers D, Santoni AL, Martins FCM, Anderson AV, et al. Intel virtualization technology. Computer 2005;38(5):48–56. doi:10.1109/MC.2005.163.

[34] Cao Y, Liu J, Miao Q, Li W. Osiris: a malware behavior capturing system implemented at virtual machine monitor layer. In: 2012 Eighth international conference on computational intelligence and security; 2012. p. 534–8. doi:10.1109/CIS.2012.126.

[35] Lengyel TK, Maresca S, Payne BD, Webster GD, Vogl S, Kiayias A. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In: Proceedings of the 30th annual computer security applications conference. ACSAC '14. New York, NY, USA: ACM; 2014. p. 386–95. ISBN 978-1-4503-3005-3. doi:10.1145/2664243.2664252.

[36] Pék G, Buttyán L. Towards the automated detection of unknown malware on live systems. In: 2014 IEEE international conference on communications (ICC); 2014. p. 847–52. doi:10.1109/ICC.2014.6883425.

[37] Rutkowska J, Tereshkin A. Isgameover anyone?. USA: Black Hat; 2007.

[38] Yan L-K, Jayachandra M, Zhang M, Yin H. V2e: Combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. SIGPLAN Not 2012;47(7):227–38. doi:10.1145/2365864.2151053.

[39] Bruening D, Duesterwald E, Amarasinghe S. Design and implementation of a dynamic optimization framework for windows. 4th ACM workshop on feedback-directed and dynamic optimization (FDDO-4); 2001.

[40] Polino M, Continella A, Mariani S, D'Alessio S, Fontana L, Gritti F, et al. Measuring and defeating anti-instrumentation-equipped malware. In: Polychronakis M, Meier M, editors. Detection of intrusions and malware, and vulnerability assessment. Cham: Springer International Publishing; 2017. p. 73–96. ISBN 978-3-319-60876-1.

[41] Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, et al. Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation. PLDI '05. New York, NY, USA: ACM; 2005. p. 190–200. ISBN 1-59593-056-6. doi:10.1145/1065010.1065034.

[42] Vasudevan A, Yerraballi R. Spike: Engineering malware analysis tools using unobtrusive binary-instrumentation. In: Proceedings of the 29th Australasian computer science conference - volume 48. ACSC '06. Darlinghurst, Australia, Australia: Australian Computer Society, Inc.; 2006. p. 311–20. ISBN 1-920682-30-9. http://dl.acm.org/citation.cfm?id=1151699.1151734.

[43] Vasudevan A, Yerraballi R. Stealth breakpoints. In: Proceedings of the 21st annual computer security applications conference. ACSAC '05. Washington, DC, USA: IEEE Computer Society; 2005. p. 381–92. ISBN 0-7695-2461-3. doi:10.1109/CSAC.2005.52.

[44] Li Z, Wang X, Liang Z, Reiter MK. Agis: Towards automatic generation of infection signatures. In: 2008 IEEE international conference on dependable systems and networks with FTCS and DCC (DSN); 2008. p. 237–46. doi:10.1109/DSN.2008.4630092.

[45] Kirat D, Vigna G, Kruegel C. Barebox: efficient malware analysis on bare-metal. In: Proceedings of the 27th annual computer security applications conference. ACSAC '11. New York, NY, USA: ACM; 2011. p. 403–12. ISBN 978-1-4503-0672-0. doi:10.1145/2076732.2076790.

[46] Grégio ARA, Filho DSF, Afonso VM, Santos RDC, Jino M, de Geus PL. Behavioral analysis of malicious code through network traffic and system call monitoring, 8059; 2011. p. 8059–8059–10 doi: 10.1117/12.883457

[47] Bulazel A, Yener B. A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In: Proceedings of the 1st reversing and offensive-oriented trends symposium. ROOTS. New York, NY, USA: ACM; 2017. ISBN 978-1-4503-5321-2. 2:1–2:21. doi: 10.1145/3150376.3150378.

[48] VirusShare.com. https://virusshare.com/ Visited on 2017-11-28;

[49] SourceForge - download, develop and publish free open source software. https://sourceforge.net/ Visited on 2019-06-07;

[50] FileHippo.com - download free software. https://filehippo.com/ Visited on 2019-06-07;

[51] Tobiyama S, Yamaguchi Y, Shimada H, Ikuse T, Yagi T. Malware detection with deep neural network using process behavior. In: Computer software and applications conference (COMPSAC), 2016 IEEE 40th annual, vol. 2. IEEE; 2016. p. 577–82.

[52] Cho IK, Kim TG, Shim YJ, Ryu M, Im EG. Malware analysis and classification using sequence alignments. Intell Autom Soft Comput 2016;22(3):371–7. doi:10.1080/10798587.2015.1118916.

[53] Damodaran A, Troia FD, Visaggio CA, Austin TH, Stamp M. A comparison of static, dynamic, and hybrid analysis for malware detection. J Comput Virol Hacking Tech 2017;13(1):1–12. doi:10.1007/s11416-015-0261-z.

[54] Gandotra E, Bansal D, Sofat S. Integrated framework for classification of malwares. In: Proceedings of the 7th International conference on security of information and networks. SIN '14. New York, NY, USA: ACM; 2014. ISBN 978-1-4503-3033-6. 417:417–417:422

[55] Total V. Virustotal-free online virus, malware and url scanner. Online: https://www virustotal com/en 2012.

[56] Guḥmundsson A. 32-bit virus threats on 64-bit windows. Tech. Rep.; Symantec; https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/32-bit-virus-threats-64-bit-windows-02-en.pdf.
[57] Chebyshev V, Sinitsyn F, Parinov D, Liskin A, Kupreev O. IT threat evolution Q1 2018. Statistics. Tech. Rep.. Kaspersky Lab; 2018. https://securelist.com/it-threat-evolution-q1-2018-statistics/85541/
[58] AVTEST. The AV-TEST security report 2015/16. Tech. Rep.; 2016. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2015-2016.pdf
[59] Oracle VM VirtualBox. https://www.virtualbox.org/ Visited on 2017-11-28;
[60] Buitinck L, Louppe G, Blondel M, Pedregosa F, Mueller A, Grisel O, et al. API design for machine learning software: experiences from the scikit-learn project. CoRR 2013;abs/1309.0238. arXiv: 1309.0238.
[61] Tian R, Islam R, Batten L, Versteeg S. Differentiating malware from cleanware using behavioural analysis. In: 2010 5th International conference on malicious and unwanted software; 2010. p. 23–30. doi:10.1109/MALWARE.2010.5665796.
[62] Firdausi I, lim C, Erwin A, Nugroho AS. Analysis of machine learning techniques used in behavior-based malware detection. In: 2010 Second international conference on advances in computing, control, and telecommunication technologies; 2010. p. 201–3. doi:10.1109/ACT.2010.33.
[63] Ahmed F, Hameed H, Shafiq MZ, Farooq M. Using spatio-temporal information in api calls with machine learning algorithms for malware detection. In: Proceedings of the 2nd ACM workshop on security and artificial intelligence. AISec '09. New York, NY, USA: ACM; 2009. p. 55–62. ISBN 978-1-60558-781-3. doi:10.1145/1654988.1655003.
[64] Miao Q, Liu J, Cao Y, Song J. Malware detection using bilayer behavior abstraction and improved one-class support vector machines. Int J Inf Secur 2016;15(4):361–79. doi:10.1007/s10207-015-0297-6.
[65] Galal HS, Mahdy YB, Atiea MA. Behavior-based features model for malware detection. J Comput VirolHacking Tech 2016;12(2):59–67. doi:10.1007/s11416-015-0244-0.
[66] Narayanan BN, Djaneye-Boundjou O, Kebede TM. Performance analysis of machine learning and pattern recognition algorithms for malware classification. In: 2016 IEEE national aerospace and electronics conference (NAECON) and Ohio innovation summit (OIS). IEEE; 2016. p. 338–42.
[67] Saleh M, Li T, Xu S. Multi-context features for detecting malicious programs. J Comput Virol Hacking Tech 2018;14(2):181–93. doi:10.1007/s11416-017-0304-8.
[68] Hansen SS, Larsen TMT, Stevanovic M, Pedersen JM. An approach for detection and family classification of malware based on behavioral analysis. In: 2016 International conference on computing, networking and communications (ICNC); 2016. p. 1–5. doi:10.1109/ICCNC.2016.7440587.
[69] Freund Y, Schapire RE. A decision-theoretic generalization of on-line learning and an application to boosting. J Comput Syst Sci 1997;55(1):119–39. doi:10.1006/jcss.1997.1504.
[70] Breiman L, Friedman J, Stone CJ, Olshen RA. Classification and regression trees. CRC press; 1984.
[71] Cortes C, Vapnik V. Support-vector networks. Mach Learn 1995;20(3):273–97. doi:10.1023/A:1022627411411.
[72] Breiman L. Random forests. Mach Learn 2001;45(1):5–32. doi:10.1023/A:1010933404324.
[73] Berlin K, Slater D, Saxe J. Malicious behavior detection using windows audit logs. In: Proceedings of the 8th ACM workshop on artificial intelligence and security. AISec '15. New York, NY, USA: ACM; 2015. p. 35–44. ISBN 978-1-4503-3826-4. doi:10.1145/2808769.2808773.
[74] Kang B, Yerima S, McLaughlin K, Sezer S. Pagerank in malware categorization. In: Proceedings of the 2015 conference on research in adaptive and convergent systems. racs. New York, NY, USA: ACM; 2015. p. 291–5. ISBN 978-1-4503-3738-0. doi:10.1145/2811411.2811514.
[75] Welch BL. The generalization of 'student's' problem when several different population variances are involved. Biometrika 1947;34(1/2):28–35. http://www.jstor.org/stable/2332510
[76] Delacre M, Lakens D, Leys C. Why psychologists should by default use welch's *t*-test instead of student's *t*-test. Int Rev Soc Psychol 2017;30(1).
[77] Ruxton GD. The unequal variance *t*-test is an underused alternative to student's *t*-test and the mann-whitney u test. Behav Ecol 2006;17(4):688–90. doi:10.1093/beheco/ark016. http://oup/backfile/content_public/journal/beheco/17/4/10.1093_beheco_ark016/2/ark016.pdf.
[78] Sikorski M, Honig A. Practical malware analysis: the hands-on guide to dissecting malicious software. No Starch Press; 2012.
[79] Data Execution Prevention. 2009. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc738483(v=ws.10).
[80] Microsoft Windows - 'nt!NtNotifyChangeDirectoryFile' Kernel Pool Memory Disclosure. https://www.exploit-db.com/exploits/42219/ Visited on 2017-07-26;
[81] Shevchenko A. Virus bulletin :: advancing malware techniques 2008. Tech. Rep.. Virus Bulletin; 2009. https://www.virusbulletin.com/virusbulletin/2009/01/advancing-malware-techniques-2008.
[82] APIMonitor.com. API monitor — spy and display win32 API calls made by applications. http://www.apimonitor.com/ Visited on 2017-07-28;
[83] Uppal D, Sinha R, Mehra V, Jain V. Malware detection and classification based on extraction of api sequences. In: 2014 International conference on advances in computing, communications and informatics (ICACCI); 2014. p. 2337–42. doi:10.1109/ICACCI.2014.6968547.
[84] Ali MAM, Maarof MA. Dynamic innate immune system model for malware detection. In: 2013 International conference on IT convergence and security (ICITCS); 2013. p. 1–4. doi:10.1109/ICITCS.2013.6717828.
[85] Chen Z-G, Kang H-S, Yin S-N, Kim S-R. Automatic ransomware detection and analysis based on dynamic api calls flow graph. In: Proceedings of the international conference on research in adaptive and convergent systems. RACS '17. New York, NY, USA: ACM; 2017. p. 196–201. ISBN 978-1-4503-5027-3. doi:10.1145/3129676.3129704.
[86] APIMon - Home. https://apimon.codeplex.com/ Visited on 2017-07-26;
[87] EasyHook. https://easyhook.github.io/ Visited on 2017-07-26;
[88] Dolgikh A, Nykodym T, Skormin V, Antonakos J, Baimukhamedov M. Colored petri nets as the enabling technology in intrusion detection systems. In: 2011 — MILCOM 2011 military communications conference; 2011. p. 1297–301. doi:10.1109/MILCOM.2011.6127481.
[89] Buster. Buster sandbox analyzer. http://bsa.isoftware.nl/, visited on 2017-07-26.
[90] Vemparala S, Di Troia F, Corrado VA, Austin TH, Stamo M. Malware detection using dynamic birthmarks. In: Proceedings of the 2016 ACM on international workshop on security and privacy analytics. IWSPA '16. New York, NY, USA: ACM; 2016. p. 41–6. ISBN 978-1-4503-4077-9. doi:10.1145/2875475.2875476.
[91] Sun M, Lin M, Chang M, Laih C, Lin H. Malware virtualization-resistant behavior detection. In: 2011 IEEE 17th international conference on parallel and distributed systems; 2011. p. 912–17. doi:10.1109/ICPADS.2011.78.
[92] Shalaginov A, Franke K. Automated intelligent multinomial classification of malware species using dynamic behavioural analysis. In: 2016 14th annual conference on privacy, security and trust (PST); 2016. p. 70–7. doi:10.1109/PST.2016.7906939.
[93] Guarnieri C, Tanasi A, Bremer J, Schloesser M. The cuckoo sandbox; 2012.
[94] Components — Cuckoo Monitor 1.3 documentation. Visited on 2017-07-28.
[95] Cho IK, Im EG. Extracting representative api patterns of malware families using multiple sequence alignments. In: Proceedings of the 2015 conference on research in adaptive and convergent systems. RACS. New York, NY, USA: ACM; 2015. p. 308–13. ISBN 978-1-4503-3738-0. doi:10.1145/2811411.2811543.
[96] Faruki P, Laxmi V, Gaur MS, Vinod P. Behavioural detection with api call-grams to identify malicious pe files. In: Proceedings of the first international conference on security of internet of things. SecurIT '12. New York, NY, USA: ACM; 2012. p. 85–91. ISBN 978-1-4503-1822-8. doi:10.1145/2490428.2490440.
[97] Qiao Y, Yang Y, He J, Tang C, Liu Z. Cbm: free, automatic malware analysis framework using api call sequences. In: Sun F, Li T, Li H, editors. Knowledge engineering and management. Berlin, Heidelberg: Springer Berlin Heidelberg; 2014. p. 225–36. ISBN 978-3-642-37832-4.
[98] Lee T, Choi B, Shin Y, Kwak J. Automatic malware mutant detection and group classification based on the n-gram and clustering coefficient. J Supercomput 2018;74(8):3489–503. doi:10.1007/s11227-015-1594-6.
[99] Lee T, Kwak J. Effective and reliable malware group classification for a massive malware environment. Int J Distrib Sens Netw 2016;12(5):4601847. doi:10.1155/2016/4601847.
[100] Fujino A, Murakami J, Mori T. Discovering similar malware samples using api call topics. In: 2015 12th annual IEEE consumer communications and networking conference (CCNC); 2015. p. 140–7. doi:10.1109/CCNC.2015.7157960.
[101] Hachinyan O. Detection of malicious software on based on multiple equations of api-calls sequences. In: 2017 IEEE conference of Russian young researchers in electrical and electronic engineering (EIConRus); 2017. p. 415–18. doi:10.1109/EIConRus.2017.7910580.
[102] Cheng JY-C, Tsai T-S, Yang C-S. An information retrieval approach for malware classification based on windows api calls. In: 2013 International conference on machine learning and cybernetics, 04; 2013. p. 1678–83. doi:10.1109/ICMLC.2013.6890868.
[103] Pirscoveanu RS, Hansen SS, Larsen TMT, Stevanovic M, Pedersen JM, Czech A. Analysis of malware behavior: Type classification using machine learning. In: 2015 International conference on cyber situational awareness, data analytics and assessment (CyberSA); 2015. p. 1–7. doi:10.1109/CyberSA.2015.7166115.
[104] Kwon I, Im EG. Extracting the representative api call patterns of malware families using recurrent neural network. In: Proceedings of the international conference on research in adaptive and convergent systems. RACS '17. New York, NY, USA: ACM; 2017. p. 202–7. ISBN 978-1-4503-5027-3. doi:10.1145/3129676.3129712.
[105] Sun B, Fujino A, Mori T. Poster: Toward automating the generation of malware analysis reports using the sandbox logs. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. CCS '16. New York, NY, USA: ACM; 2016. p. 1814–16. ISBN 978-1-4503-4139-4. doi:10.1145/2976749.2989064.
[106] Gandotra E, Bansal D, Sofat S. Zero-day malware detection. In: 2016 sixth international symposium on embedded computing and system design (ISED); 2016. p. 171–5. doi:10.1109/ISED.2016.7977076.
[107] Dhammi A, Singh M. Behavior analysis of malware using machine learning. In: Contemporary computing (IC3), 2015 eighth international conference on. IEEE; 2015. p. 481–6.
[108] Fraley JB, Figueroa M. Polymorphic malware detection using topological feature extraction with data mining. In: SoutheastCon 2016; 2016. p. 1–7. doi:10.1109/SECON.2016.7506685.
[109] Pektaş A, Acarman T, Falcone Y, Fernandez J. Runtime-behavior based malware classification using online machine learning. In: 2015 World congress on internet security (WorldCIS); 2015. p. 166–71. doi:10.1109/WorldCIS.2015.7359437.

[110] Zhang Y, Rong C, Huang Q, Wu Y, Yang Z, Jiang J. Based on multi-features and clustering ensemble method for automatic malware categorization. In: 2017 IEEE trustcom/BigDataSE/ICESS; 2017. p. 73–82. doi:10.1109/Trustcom/BigDataSE/ICESS.2017.222.
[111] Lim C, Ramli K. Mal-one: A unified framework for fast and efficient malware detection. In: 2014 2nd International conference on technology, informatics, management, engineering environment; 2014. p. 1–6. doi:10.1109/TIME-E.2014.7011581.
[112] Wüchner T, Ochoa M, Lovat E, Pretschner A. Generating behavior-based malware detection models with genetic programming. In: 2016 14th ANNUAL CONFERENCE ON PRIVACY, SECURITY AND TRUSt (PST); 2016. p. 506–11. doi:10.1109/PST.2016.7907008.
[113] Bazzi A, Onozato Y. Ids for detecting malicious non-executable files using dynamic analysis. In: 2013 15th Asia-Pacific network operations and management symposium (APNOMS); 2013. p. 1–3.
[114] Kim D, Majlesi-Kupaei A, Roy J, Anand K, ElWazeer K, Buettner D, et al. Dynodet: Detecting dynamic obfuscation in malware. In: Polychronakis M, Meier M, editors. Detection of intrusions and malware, and vulnerability assessment. Cham: Springer International Publishing; 2017. p. 97–118. ISBN 978-3-319-60876-1.
[115] Baychev Y, Bilge L. Spearphishing malware: Do we really know the unknown?. In: Giuffrida C, Bardin S, Blanc G, editors. Detection of intrusions and malware, and vulnerability assessment. Cham: Springer International Publishing; 2018. p. 46–66. ISBN 978-3-319-93411-2.
[116] Kolosnjaji B, Zarras A, Lengyel T, Webster G, Eckert C. Adaptive semantics-aware malware classification. In: Proceedings of the 13th international conference on detection of intrusions and malware, and vulnerability assessment - volume 9721. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag; 2016. p. 419–39. ISBN 978-3-319-40666-4. doi:10.1007/978-3-319-40667-1_21.
[117] Wüchner T, Ochoa M, Pretschner A. Robust and effective malware detection through quantitative data flow graph metrics. In: Proceedings of the 12th international conference on detection of intrusions and malware, and vulnerability assessment - volume 9148. DIMVA 2015. Berlin, Heidelberg: Springer-Verlag; 2015. p. 98–118. ISBN 978-3-319-20549-6. doi:10.1007/978-3-319-20550-2_6.
[118] Sharma A, Gandotra E, Bansal D, Gupta D. Malware capability assessment using fuzzy logic. Cybern Syst 2019;50(4):323–38. doi:10.1080/01969722.2018.1552906.
[119] Ijaz M, Durad MH, Ismail M. Static and dynamic malware analysis using machine learning. In: 2019 16th International bhurban conference on applied sciences and technology (IBCAST); 2019. p. 687–91. doi:10.1109/IBCAST.2019.8667136.
[120] Thebeyanthan K, Achsuthan M, Ashok S, Vaikunthan P, Senaratne AN, Abeywardena KY. E-secure: An automated behavior based malware detection system for corporate e-mail traffic. In: Arai K, Kapoor S, Bhatia R, editors. Intelligent computing. Cham: Springer International Publishing; 2019. p. 1056–71. ISBN 978-3-030-01177-2.
[121] Kakisim AG, Nar M, Carkaci N, Sogukpinar I. Analysis and evaluation of dynamic feature-based malware detection methods. In: Lanet J-L, Toma C, editors. Innovative security solutions for information technology and communications. Cham: Springer International Publishing; 2019. p. 247–58. ISBN 978-3-030-12942-2.
[122] Shiva Darshan SL, Jaidhar CD. Empirical study on features recommended by lsvc in classifying unknown windows malware. In: Bansal JC, Das KN, Nagar A, Deep K, Ojha AK, editors. Soft computing for problem solving. Singapore: Springer Singapore; 2019. p. 577–90. ISBN 978-981-13-1595-4.
[123] Hsiao S, Yu F. Malware family characterization with recurrent neural network and ghsom using system calls. In: 2018 IEEE International conference on cloud computing technology and science (CloudCom); 2018. p. 226–9. doi:10.1109/CloudCom2018.2018.00051.
[124] Jamalpur S, Navya YS, Raja P, Tagore G, Rao GRK. Dynamic malware analysis using cuckoo sandbox. In: 2018 Second international conference on inventive communication and computational technologies (ICICCT); 2018. p. 1056–60. doi:10.1109/ICICCT.2018.8473346.
[125] Tungjitviboonkun T, Suttichaya V. Complexity reduction on api call sequence alignment using unique api word sequence. In: 2017 21st international computer science and engineering conference (ICSEC); 2017. p. 1–5. doi:10.1109/ICSEC.2017.8443930.
[126] Takeuchi Y, Sakai K, Fukumoto S. Detecting ransomware using support vector machines. In: Proceedings of the 47th international conference on parallel processing companion. ICPP '18. New York, NY, USA: ACM; 2018. ISBN 978-1-4503-6523-9. 1:1–:1.6. doi: 10.1145/3229710.3229726.
[127] Babenko L, Kirillov A. Development of method for malware classification based on statistical methods and an extended set of system calls data. In: proceedings of the 11th international conference on security of information and networks. SIN '18. New York, NY, USA: ACM; 2018. ISBN 978-1-4503-6608-3. 8:1–:8.6. doi: 10.1145/3264437.3264478.
[128] Willems C, Holz T, Freiling F. Toward automated dynamic malware analysis using cwsandbox. IEEE Secur Priv 2007;5(2):32–9. doi:10.1109/MSP.2007.45.
[129] Rieck K, Trinius P, Willems C, Holz T. Automatic analysis of malware behavior using machine learning. J Comput Secur 2011;19(4):639–68. http://dl.acm.org/citation.cfm?id=2011216.2011217.
[130] Qiao Y, Yang Y, Ji L, He J. Analyzing malware by abstracting the frequent itemsets in api call sequences. In: 2013 12th IEEE international conference on trust, security and privacy in computing and communications; 2013. p. 265–70. doi:10.1109/TrustCom.2013.36.
[131] Qiao Y, He J, Yang Y, Ji L, Tang C. A lightweight design of malware behavior representation. In: Trust, security and privacy in computing and communications (TrustCom), 2013 12th IEEE international conference on. IEEE; 2013. p. 1607–12.
[132] Li HJ, Tien C, Tien C, Lin C, Lee H, Jeng AB. Aos: an optimized sandbox method used in behavior-based malware detection. In: 2011 International conference on machine learning and cybernetics, 1; 2011. p. 404–9. doi:10.1109/ICMLC.2011.6016683.
[133] Goebel J, Holz T, Willems C. Measurement and analysis of autonomous spreading malware in a university environment. In: Proceedings of the 4th international conference on detection of intrusions and malware, and vulnerability assessment. DIMVA '07. Berlin, Heidelberg: Springer-Verlag; 2007. p. 109–28. ISBN 978-3-540-73613-4. doi:10.1007/978-3-540-73614-1_7.
[134] Rieck K, Holz T, Willems C, Düssel P, Laskov P. Learning and classification of malware behavior. In: Proceedings of the 5th international conference on detection of intrusions and malware, and vulnerability assessment. DIMVA '08. Berlin, Heidelberg: Springer-Verlag; 2008. p. 108–25. ISBN 978-3-540-70541-3. doi:10.1007/978-3-540-70542-0_6.
[135] Deviare API | Hook Nektra - fast custom software development company. (visited on 2017-09-30).
[136] Park Y, Reeves D, Mulukutla V, Sundaravel B. Fast malware classification by automated behavioral graph matching. In: proceedings of the sixth annual workshop on cyber security and information intelligence research. CSIIRW '10; New York, NY, USA: ACM. 2010, p. 978-1-4503-0017-9,45:1–45:4. 10.1145/1852666.1852716
[137] Naval S, Laxmi V, Rajarajan M, Gaur MS, Conti M. Employing program semantics for malware detection. IEEE Trans Inf ForensSecur 2015;10(12):2591–604. doi:10.1109/TIFS.2015.2469253.
[138] Gupta S, Sharma H, Kaur S. Malware characterization using windows api call sequences. In: Carlet C, Hasan MA, Saraswat V, editors. Security, privacy, and applied cryptography engineering. Cham: Springer International Publishing; 2016. p. 271–80. ISBN 978-3-319-49445-6.
[139] Malpimp: Advanced API Tracing Tool. http://securityxploded.com/malpimp.php Visited on 2017-07-26;
[140] Fan C, Hsiao H, Chou C, Tseng Y. Malware detection systems based on api log data mining. In: 2015 IEEE 39th annual computer software and applications conference, 3; 2015. p. 255–60. doi:10.1109/COMPSAC.2015.241.
[141] Inoue D, Yoshioka K, Eto M, Hoshizawa Y, Nakao K. Automated malware analysis system and its sandbox for revealing malware's internal and external activities. IEICE Trans Inf Syst 2009;E92.D(5):945–54. doi:10.1587/transinf.E92.D.945.
[142] Kasama T, Yoshioka K, Inoue D, Matsumoto T. Malware detection method by catching their random behavior in multiple executions. In: 2012 IEEE/IPSJ 12th international symposium on applications and the internet; 2012. p. 262–6. doi:10.1109/SAINT.2012.49.
[143] NtTrace. http://www.howzatt.demon.co.uk/NtTrace/ Visited on 2017-07-26;
[144] Jang J, Woo J, Mohaisen A, Yun J, Kim HK. Mal-netminer: malware classification approach based on social network analysis of system call graph. CoRR 2016;abs/1606.01971. arXiv: 1606.01971.
[145] IntellectualHeaven StraceNT - strace for windows. http://intellectualheaven.com/default.asp?BH=StraceNT Visited on 2017-07-26;
[146] Nair VP, Jain H, Golecha YK, Gaur MS, Laxmi V. Medusa: Metamorphic malware dynamic analysis usingsignature from api. In: Proceedings of the 3rd international conference on security of information and networks. SIN '10. New York, NY, USA: ACM; 2010. p. 263–9. ISBN 978-1-4503-0234-0. doi:10.1145/1854099.1854152.
[147] Patanaik CK, Barbhuiya FA, Nandi S. Obfuscated malware detection using api call dependency. In: Proceedings of the first international conference on security of internet of things. SecurIT '12. New York, NY, USA: ACM; 2012. p. 185–93. ISBN 978-1-4503-1822-8. doi:10.1145/2490428.2490454.
[148] Wang X, Yu W, Champion A, Fu X, Xuan D. Detecting worms via mining dynamic program execution. In: 2007 Third international conference on security and privacy in communications networks and the workshops - SecureComm 2007; 2007. p. 412–21. doi:10.1109/SECCOM.2007.4550362.
[149] Fukushima Y, Sakai A, Hori Y, Sakurai K. A behavior based malware detection scheme for avoiding false positive. In: 2010 6th IEEE workshop on secure network protocols; 2010. p. 79–84. doi:10.1109/NPSEC.2010.5634444.
[150] Blokhin K, Saxe J, Mentis D. Malware similarity identification using call graph based system call subsequence features. In: Proceedings of the 2013 IEEE 33rd international conference on distributed computing systems workshops. ICDCSW '13. Washington, DC, USA: IEEE Computer Society; 2013. p. 6–10. ISBN 978-1-4799-3248-1. doi:10.1109/ICDCSW.2013.55.
[151] Liu S, Huang H, Chen Y. A system call analysis method with mapreduce for malware detection. In: 2011 IEEE 17th international conference on parallel and distributed systems; 2011. p. 631–7. doi:10.1109/ICPADS.2011.17.
[152] Yang Y, Cai Z, Mao W, Yang Z. Identifying intrusion infections via probabilistic inference on bayesian network. In: Proceedings of the 12th international conference on detection of intrusions and malware, and vulnerability assessment - volume 9148. DIMVA 2015. Berlin, Heidelberg: Springer-Verlag; 2015. p. 307–26. ISBN 978-3-319-20549-6. doi:10.1007/978-3-319-20550-2_16.
[153] Snihurov A, Shulhin O, Balashov V. Experimental studies of ransomware for developing cybersecurity measures. In: 2018 International scientific-practical conference problems of infocommunications. science and technology (PIC S T); 2018. p. 691–5. doi:10.1109/INFOCOMMST.2018.8632153.

[154] Yin H, Song D. Temu: Binary code analysis via whole-system layered annotative execution. Tech. Rep. UCB/EECS-2010-3. EECS Department, University of California, Berkeley; 2010. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-3.html.

[155] Xu Z, Chen L, Gu G, Kruegel C. Peerpress: utilizing enemies' p2p strength against them. In: Proceedings of the 2012 ACM conference on computer and communications security. CCS '12. New York, NY, USA: ACM; 2012. p. 581–92. ISBN 978-1-4503-1651-4. doi:10.1145/2382196.2382257.

[156] Ugarte-Pedrero X, Balzarotti D, Santos I, Bringas PG. Rambo: Run-time packer analysis with multiple branch observation. In: Caballero J, Zurutuza U, Rodríguez RJ, editors. Detection of intrusions and malware, and vulnerability assessment. Cham: Springer International Publishing; 2016. p. 186–206. ISBN 978-3-319-40667-1.

[157] Bayer U, Kruegel C, Kirda E. Anubis: analyzing unknown binaries; 2009.

[158] Egele M, Scholte T, Kirda E, Kruegel C. A survey on automated dynamic malware-analysis techniques and tools. ACM Comput Surv (CSUR) 2008;44(2). 6:1–6:42 doi: 10.1145/2089125.2089126.

[159] Lindorfer M, Di Federico A, Maggi F, Comparetti PM, Zanero S. Lines of malicious code: insights into the malicious software industry. In: Proceedings of the 28th annual computer security applications conference. ACSAC '12. New York, NY, USA: ACM; 2012. p. 349–58. ISBN 978-1-4503-1312-4. doi:10.1145/2420950.2421001.

[160] Kolbitsch C, Kirda E, Kruegel C. The power of procrastination: detection and mitigation of execution-stalling malicious code. In: Proceedings of the 18th ACM conference on computer and communications security. CCS '11. New York, NY, USA: ACM; 2011. p. 285–96. ISBN 978-1-4503-0948-6. doi:10.1145/2046707.2046740.

[161] Kirat D, Vigna G. Malgene: Automatic extraction of malware analysis evasion signature. In: Proceedings of the 22Nd ACM SIGSAC conference on computer and communications security. CCS '15. New York, NY, USA: ACM; 2015. p. 769–80. ISBN 978-1-4503-3832-5. doi:10.1145/2810103.2813642.

[162] Graziano M, Canali D, Bilge L, Lanzi A, Balzarotti D. Needles in a haystack: mining information from public dynamic analysis sandboxes for malware intelligence. In: 24th USENIX security symposium (USENIX Security 15). Washington, D.C.: USENIX Association; 2015. p. 1057–72. ISBN 978-1-931971-232. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/graziano.

[163] Kolbitsch C, Comparetti PM, Kruegel C, Kirda E, Zhou X, Wang X. Effective and efficient malware detection at the end host. In: Proceedings of the 18th conference on usenix security symposium. SSYM'09. Berkeley, CA, USA: USENIX Association; 2009. p. 351–66. http://dl.acm.org/citation.cfm?id=1855768.1855790.

[164] WinAPIOverride: free advanced API monitor, spy or override API or exe internal functions. http://jacquelin.potier.free.fr/winapioverride32/index.php Visited on 2017-10-23;

[165] Salehi Z, Sami A, Ghiasi M. Using feature generation from api calls for malware detection. Comput Fraud Secur 2014;2014(9):9–18. doi:10.1016/S1361-3723(14)70531-7.

[166] Salehi Z, Ghiasi M, Sami A. A miner for malware detection based on api function calls and their arguments. In: The 16th CSI international symposium on artificial intelligence and signal processing (AISP 2012); 2012. p. 563–8. doi:10.1109/AISP.2012.6313810.